第一章 计 算

——它不仅仅是一位匠人

数学不等于计算，但数学的最早来源是计算. 它诞生于人类生产实践中对计算的需要，数字本身就是这样产生的. 有人考证过数字"二"的出现，据说这表示了双手各拿一件物品. 为了表示"三"，除双手拿物品之外，还要把第三件物品放在自己的脚边，"三"的特征就是举起双手和指定一只脚. "四"可以用两只手和两只脚表示，等等. 此外，手指也是表示数量的最早工具. 为什么世界最通用的是十进制，而不是九进制或十一进制？据说这和人有十个指头有关. 在长期的发展中，人们逐渐抛弃了以具体物件表示数的做法，慢慢地抽象出数的概念.

记数术可以说是最早的数学，从屈指记数，结绳计数，到在木棍或骨片上刻符记数，以及书写文字记数，经历了漫长的年代. 据史书记载，世界上几个最早的记数系统的出现年代大致如下：

1. 古埃及象形数字(公元前 3400 年左右).
2. 巴比伦楔形数字(公元前 2400 年左右).
3. 中国甲骨文数字(公元前 1600 年左右).

VISIT…

LANZAROTE
Caliente.com

4. 中国算筹(公元前 500 年左右).

十进制数的出现是一个伟大的事件，它是一种位置记数法，其中每个数字的含义与数字所处的位置有关，$19\neq91$. 十进制技术的核心是使用数字零，不然，91 和 901，910 等就无法区分了. 一般认为公元 600 年前后在古印度数学中明确形成了方便的十进制记数，包括零的使用. 这套方法后来传到阿拉伯地区，以致一直以阿拉伯数字的称呼流传后世. 实际上，世界上最早的十进制数表示法是中国的算筹记数法，其中包括了零的应用. 用算筹表示数字，有纵横两种方式：

纵式：	𝍠	𝍡	𝍢	𝍣	𝍤	𝍥	𝍦	𝍧	𝍨	空格
横式：	𝍩	𝍪	𝍫	𝍬	𝍭	𝍮	𝍯	𝍰	𝍱	空格
含义：	1	2	3	4	5	6	7	8	9	0

使用时规定个位数用纵式表示，十位数以上交叉使用，例如，314159265 表示为

𝍢 𝍩 𝍣 𝍩 𝍤 𝍱 𝍡 𝍮 𝍤

几何计算也来自生产实践. 古埃及的尼罗河水每年都要泛滥，淹没庄稼，冲毁土地，这给土地的管理带来一些问题. 据史书记载，埃及第十九王朝的法老拉美西斯第二(约公元前 1300 年)，把土地划分成同样大小的正方形，并分配给居民，按分配土地的面积收租. 如果河水泛滥时冲毁了部分土地，便要重新丈量土地，根据土地损失情况核减租赋，有时甚至还要重新分配土地，这样. 以面积测量为特征的最早的几何学就诞生了.

由于计算是生产实践的需要，因而在它发展的早期主要是一种技术. 例如对圆周率 π，一开始没有明确的计算公式，π 的值

能算到多精确，全凭数学家的本事. 在古巴比伦数学中，只知道 π 近似于 3. 公元前 6 世纪，印度数学家算到 π 近似于3.162. 公元 300 年左右，中国数学家刘徽用割圆术把 π 的近似值改进为 3.1416. 一百多年以后，祖冲之又算得 π 的值在 3.1415926 与 3.1415927 之间. 西方数学家得到同样精确的结果则已是一千年以后的事了.

作为一种技术，有时难免要发生保密、泄密和窃密的问题. 三次代数方程式的解法是 16 世纪的意大利数学家塔尔塔利亚发明的. 另一位意大利数学家菲奥尔声称自己也找到了三次方程的解法，两人相约在米兰公开比赛，结果菲奥尔大败. 不久，卡尔达诺向塔尔塔利亚请教该法，遭到拒绝. 卡尔达诺遂许愿推荐他担任西班牙炮兵顾问，并发誓永不泄密，以此骗取了三次方程式解法，并在六年后违约公布于自己写的书中. 后世传称三次方程式的解法为卡尔达诺算法，这是不公正的. 然而这件事也反映了古代某些计算技术正像我国的祖传秘方那样，属于个人的秘密.

重视实际的计算方法是我国古代数学的一大特点，在古籍中往往称之以“术”. 如上面提到的以极限方法求圆周率的割圆术；在解算术问题时使用变元方法的天元术；以及求解一次同余方程组的大衍求一术等. 与我国的传统不同，古希腊的数学家们发展出一套严密的逻辑演绎方法，不但有亚里斯多德的形式逻辑，还有欧几里德的几何体系，这是数学史上另一条重要的发展线索. 当然，逻辑方法本身还不是计算，我们在以后将会讲到，只有当数理逻辑问世以后，才有了以计算方法处理逻辑问题的手段.

也正是从希腊人开始，出现了对计算的根本问题——什么是可计算的，什么是不可计算的——的研究，其成果往往是某些

问题类的不可计算性，由此又导出一些新的、更富有意义的研究领域. 要知道，指出某些问题是不可计算的，其意义决不亚于给出可计算问题的计算方法，这在数学史上屡见不鲜. 一个著名的事件是公元前 400 年希腊数学家希帕索斯发现直角三角形的直角边与斜边之长是不可通约的，也就是说，这两条边的商不能用有理数表示. 虽然希帕索斯因此而被抛进大海处死，但由此而产生的无理数概念却为数学带来了新的繁荣. 另一个著名的例子是用根式求解一般五次方程的问题，挪威数学家阿贝尔在 1824 年指出这是不可能的. 阿贝尔的卓越成果推动人们去寻求五次以上方程有根式解的一般条件. 五年以后，这个问题被法国数学家伽罗瓦解决，从而又导致一门新的数学分支——群论的产生. 作为伽罗瓦的发现的一个推论，人们知道了用直尺和圆规三等分任意角是不可能的.

但这些都是关于某类具体问题求解的可能性或不可能性的研究. 至于最一般的问题类的可计算性研究，则直到 20 世纪 30 年代才结出硕果. 图灵、丘奇、哥德尔、波斯特等人陆续提出了一批计算模型，并称这些模型为用算法方法解决问题的极限. 即：凡是能用算法方法解决的问题，也一定能用这些计算模型解决. 反之，这些计算模型解决不了的问题，任何算法也休想解决. 本书第二至第六章列出了五种这样的计算模型，实际的计算模型体现于现代的电子计算机及其程序设计语言上，第七章对此作了简要介绍.

这里说到了算法，需要解释一下什么是算法. 算法一词来自古代阿拉伯一本数学名著的书名，它指的是一种计算过程，具有如下的性质：

1. 通用性. 即适用于某一类问题中的所有个体，而不是只用来解决一个具体问题.

2. 能行性. 即应有明确的步骤一步一步地引导计算的进行.

3. 机械性. 即每个步骤都是机械的、定死的，不需要计算者临时动脑筋.

4. 有限性. 至少对某些输入数据，算法应在有限多步内结束，并给出计算结果. 我们称算法对这些输入数据是有定义的.

5. 离散性. 算法的输入数据及输出数据都应是离散的符号(或称字母，其中也包括数字).

虽然人们已经相信，在上述算法定义的意义上，不存在第二到第七章的计算模型不能计算、而其它模型能计算的问题，但是人们也已了解到，确实存在着任何计算模型都计算不了的问题，第八章讨论这个十分重要的不可计算性和不可判定性问题. 就像告诫人们三等分角不可能一样，这方面的研究成果可使人们避免无效的劳动. 与此相关的问题是形式系统的一致性和完备性问题，这在第九章中探讨. 总之，第八、九两章说的是现实世界计算能力的极限.

计算模型的计算能力还受着另一个因素的限制，那就是计算的复杂性. 通俗地说，可称之为计算的繁复程度. 对此，人们首先想到的是计算所需的步数，但这只是复杂性的一种，称为时间复杂性. 通常用计算所需的各种开销之大小衡量计算复杂性，时间是一种开销，空间(可想象为算题用的草稿纸)也是一种开销，还可以定义其它的开销. 第十章研究抽象计算模型的计算复杂性，第十一章研究具体的算法类中被认为最难的一类——NP完备类的复杂性，这两方面都已有极丰富的研究成果.

有关计算的另一个大问题是计算的语义问题. 通俗地讲即是：怎样保证计算的结果是我们所需要的？即如果一个算法用两种形式表示出来，一种是说明性的(例如求两个数的最大公约数)，另一种是过程性的(例如欧几里德辗转相除法)，如何才能

判断:过程性的算法达到了说明性算法的要求?计算的语义问题是一个极大的问题,本书只能在最后一章触及它的很小一点皮毛,供读者品味.

第二章　图灵机

——跳不出的如来佛手心

计算的能行性和可构造性研究的最著名产物，也许是图灵机了. 图灵是一位英国数学家，生于 1912 年 6 月 23 日，卒于 1954 年 6 月 7 日，只活了 42 岁，然而却在数学和计算理论方面作出了卓越的贡献. 他在 1937 年发表的《关于可计算的数及其对判定问题的应用》一文中提出了一个非常重要的关于计算的数学模型，后世称之为图灵机. 其重要性在于：这是一个能行的计算模型，它的原理非常之简单，然而却能计算一切能行可计算的问题类，因此它的功能不会弱于任何其它的能行计算模型. 当然，这个结论从未被严格证明过，而且，它是证明不了的. 因为图灵机是一个严格的数学模型，而所谓可计算的问题类则是一个用语言描述的模糊概念. 我们只能证明一个精确的数学概念等价于另一个精确的数学概念，却无法“证明”一个模糊的概念等价于一个精确的概念. 不过，从图灵机创立以来的无数事实使人们相信这一点是对的. 还从未有人发现过一个为图灵机计算不了的问题类，就像孙悟空跳不出如来佛的手心一样. 现在，数学家们不但不怀疑这一点，而且把“图灵机可计算”作为能行可计

算的代名词. 各种各样的关于计算的数学模型，均以能被证明和图灵机等价作为它们具有最高计算能力的标志. 图灵的论文发表不到十年，现实的电子计算机就问世了. 计算机科学家们公认图灵的思想为电子计算机的诞生奠定了理论基础，图灵因此而获得了一项殊荣，目前国际上授予计算机科学家的最高学术成就奖就是以他的名字命名的. 所以，一切对计算的数学理论感兴趣的人都应该认识和了解图灵机.

图灵机的计算在一条带子上进行，这条带子在两个方向均为无穷长(可以把它想象成解析几何中的 x 轴). 带子上划分为无穷多个格(可以把它想象为 x 轴上长度为 1 的区间). 带子上方有一个沿带子方向来回移动的读写头(读写头和带子的关系有点象录音机中磁头和磁带的关系). 计算通过读写头的移动和读写来完成. 为了控制读写头的这些操作，每个图灵机有一个状态集 $\{q_i\}$，其中包括一个开始状态和一个结束状态. 它还有一组符号 $\{s_i\}$，其中包括一个空白符号. 此外，它还有一个控制函数，该函数根据图灵机所处的当前状态和读写头所读到的当前符号决定图灵机的下一步操作，其中每一步操作包括三件事：第一，把某个符号写到读写头当前正“注视”的那个格上，以取代原来的符号. 第二，读写头左移一格或右移一格或不移动. 第三，用某个状态取代当前的状态，使图灵机进入一个新状态. 这个控制函数可以表为：

$$(\text{状态}, \text{符号}) \longrightarrow (\text{写符号}, \text{移动}, \text{状态}) \qquad (2.1)$$

顺序做完这三件事，图灵机的一个工作周期就告结束. 如果新状态不是结束状态，则可以进入下一个工作周期，否则图灵机停机，计算任务宣告完成. 所以结束状态也叫停机状态. 图灵机停机以后，带子上的内容就是它的输出. 对于图灵机的工作过程，人们可以给予不同的解释.

第一，把图灵机作为识别器，即判断带子上的初始内容（看作图灵机的输入）是否属于某个集合（例如是否是一个语法正确的英语句子）. 如果图灵机能够停机，并且输出某个指定的符号（例如 yes），则表明输入内容确实属于指定的集合. 否则，如果输出的不是指定的符号，或者图灵机根本不停机，则表明输入内容不属于指定集合，它被拒绝.

第二，把图灵机作为生成器，对于给定的输入集合，考察它的输出集合，并研究当输入集合满足某种性质时，输出集合满足什么性质，这是研究图灵机的生成能力.

第三，把图灵机作为计算器，对于指定的计算功能（什么样的输入应该对应什么样的输出），设计恰当的图灵机，使它具备这种计算功能，这样的图灵机相当于一个函数. 对于某些输入值，图灵机可能不会停机，这相当于该函数在此处无定义. 因此，图灵机可以定义一个部分函数（即并非处处有定义的函数）. 以后我们将会看到，它的能力相当于部份递归函数.

对图灵机工作过程的以上三种解释，实际上只反映了人在感觉上的不同，它们本质上是一致的，归根结底，都是一种计算. 下面我们举一个最简单的例子——设计一个会做加法的图灵机，这个图灵机只有两种符号：0 和 1，其中 0 代表空白符号. 它计算的对象——正整数，用一连串的 1 表示，1 的个数等于此正整数的大小加一. 有待相加的两个数之间用一个空白符号隔开，相加后的数也用一连串的 1 表示. 该图灵机的控制函数可用第 10 面的表列出.

其中 L 表示左移. R 表示右移，开始的时候，读写头“注视”着左边那个数的第一个符号 1，状态为 0. 查表可知动作应为 $1R1$，即在原地重写符号 1，右移一格，并进入状态 1，如果读写头此时注视的当前符号仍为 1，则查表知动作为 $1R1$，与刚才一

状态 \ 符号	0	1
0		$1R1$
1	$1R2$	$1R1$
2	$0L3$	$1R2$
3		$0L4$
4		$0L5$
5	$0R6$	$1L5$
6	STOP	

样，并停留在状态 1 内不变. 这个过程继续不断，直至读写头移到两数间的空格，此时动作为 $1R2$，即把空格改为符号 1 并继续右移，状态变为 2. 一直移到第二个数右边的空格. 此时保留该空格，往左移，并进入状态 3，接着连续抹掉两个 1(把它们改成空格符号)，并一直往左移，直至移到此数左边的空格，再右移回来. 因为图灵机要求计算结束时，读写头对准输出数据左面的第一个符号.

形象地说，加法不过是把带子上的两个数拼接在一起，但若用图灵机严格描述，就必须经过上面那些复杂的步骤，有兴趣的读者不妨自己设计一个做乘法的图灵机.

图灵机有许多变种，其中有的与图灵机有相同的计算能力，有的在计算能力上弱于图灵机，下面我们介绍其中的几种.

首先是多道图灵机. 它与标准图灵机的区别在于：带子分成有限多个道，每个道相当于一个标准图灵机的带子. 我们在前面曾经把图灵机的带子比做 x 轴(即直线 $y=0$)，多道图灵机的带

子好比一组平行于 x 轴的直线($y=0,1,\cdots,n-1$),在同一个垂直位置上,各直线格内的内容可以不一样,但读写头只有一个.下表是一个三道图灵机的例子,它的控制函数可以表为:

(状态,上符,中符,下符)──→(写上符,写中符,写下符,移动,状态)　　(2.2)

⇓读写头

……	0	0	×	0	√	#	#	×	√	#	0	0	……
……	√	×	#	√	0	#	×	×	√	0	√	#	……
……	#	√	0	#	0	√	√	0	×	0	×	×	……

很容易证明如下的定理:

定理 2.1　多道图灵机和标准(即单道)图灵机有相同的计算能力.

证:首先,多道图灵机可以模拟单道图灵机,因为对任意的单道图灵机和任意正整数 $n>1$,可以构造如下的 n 道图灵机:除了第一道以外,其它道上的格内全为空白符号,并且在图灵机操作时在第一道以外的各道只写空白符号.至于第一道的初始内容以及往第一道上写的内容,则和指定的单道图灵机完全一样.不难看出,这样的多道图灵机(见下表)与指定的单道图灵机有相同功能.

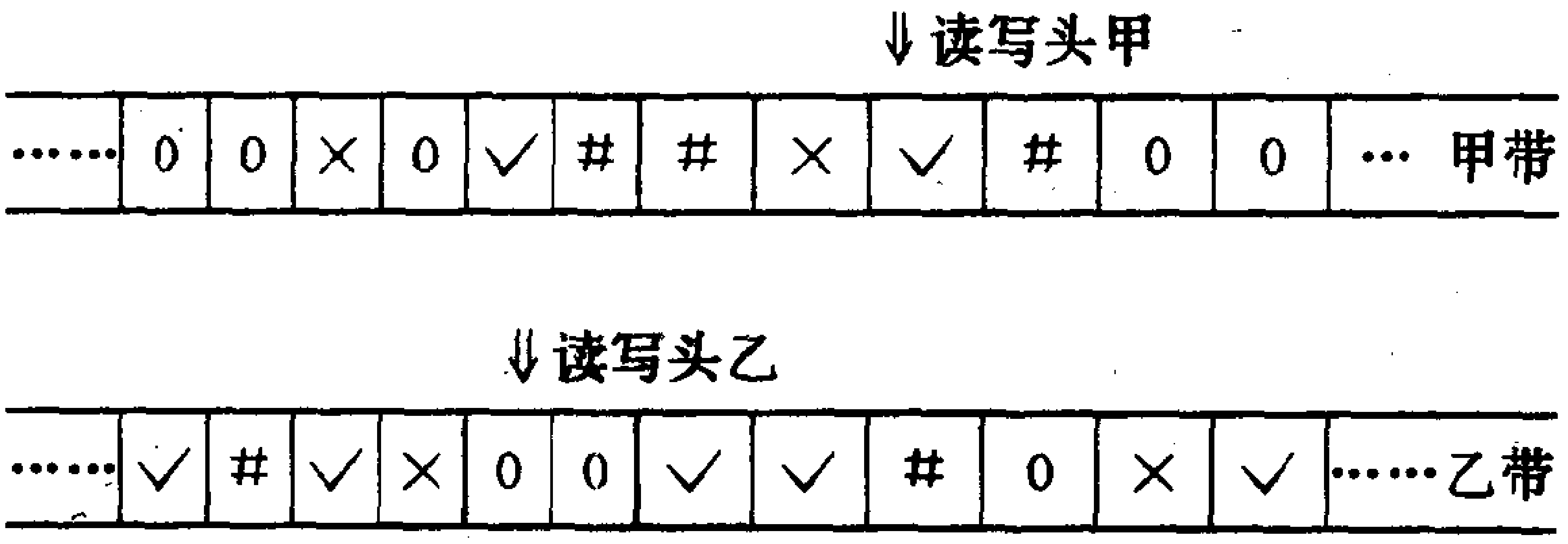
⇓读写头

	0	0	×	0	√	#	#	×	√	#	0	0	
	0	0	0	0	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	0	0	0	0	

另一方面,单道图灵机也可以模拟多道图灵机.假设给定一

个 n 道图灵机，$n>1$. 它共有 m 种不同的符号，则在每个垂直位置上，各格内不同符号的组合共有 m^n 种可能性. 为此，我们只需构造一个具有 m^n 种不同符号的单道图灵机，并在单道图灵机的符号和多道图灵机的符号组合之间建立起一一对应关系. 这样，多道图灵机的辨认 n 个符号和书写 n 个符号，完全可以用单道图灵机的辨认一个符号和书写一个符号代替，两者功能也就一致了.

第二个变种是多带图灵机. 一个 n 带图灵机有 n 根带子，每根带子上有一个读写头. 各读写头的移动是彼此独立的. 下图是一个双带图灵机的例子.



双带图灵机的控制函数可以表为：

(状态，上符，下符)⟶(写上符，写下符，移上头，移下头，状态)　　(2.3)

多带图灵机的结构复杂性显然高于多道图灵机，因为它的多个读写头是互相独立移动的. 不过我们仍然有下列定理：

定理 2.2　多带图灵机和标准图灵机有相同的计算能力.

证：为了节省篇幅，此处只给出证明的梗概. 由于我们已经有了定理 2.1，所以我们只需证明多道图灵机和多带图灵机有相同的计算能力就行了.

首先，给定一个 n 道图灵机，$n>1$，不难构造一个 n 带图灵

机来模拟它，因为我们只需令 n 带图灵机的 n 个读写头永远位于同一个垂直位置上，其余的数据和操作可以全都照抄 n 道图灵机的，这就把一个 n 带图灵机的功能设计得和给定的 n 道图灵机一样了.

现在假设给定一个 n 带图灵机，模拟 n 带图灵机的困难在于 n 道图灵机只有一个读写头. 我们可以用一个读写头来回移动以完成所有读写任务，但是怎样才能记住另外 $n-1$ 个读写头的位置呢?我们假设任意两个不同的带上的符号集之交是空集，然后把符号数增加一倍，即对每个符号 a，增加一个符号 a'，一条带子上某个格子内有符号 a 表示该格子的内容是 a，且当前读写头不位于此处. 而格子内有符号 a' 表示该格子内容是 a 且当前读写头正位于此处. 通过正确构造控制函数可使每条带上每个时刻有且仅有一个 a' 类符号.

由于多道图灵机只有一个读写头，在 n 条带上的读写只能逐个进行，根据前面控制函数的定义，多带图灵机的每条带上的操作依赖于所有带上的当前符号. 多道图灵机在第 i 条带上读写后要移动读写头，此时读写头注视的内容已经不是原来的内容，不能作为第 j 条带($j>i$)操作的依据，解决这个问题的办法是把当前符号也记进状态里去，多带图灵机的每个状态 S，在多道图灵机中对应于 $\prod_{i=1}^{n} \mathrm{no}S(i)$ 个不同的状态，其中 $\mathrm{no}S(i)$ 是第 i 条带上符号的种类数，新状态取 $(a_{1i_1}, a_{2i_2}, \cdots, a_{ni_n}, S)$ 的形式，其中 a_{ij} 表示第 i 条带上的第 j 种符号.

于是，多道图灵机的控制函数可以这样构造：开始时令每条带上数据的最左一个符号为 a' 类符号，并令初始状态为 $(a'_{1i_1}, a'_{2i_2}, \cdots, a'_{ni_n}, S_0)$，其中诸 a'_{ij} 为第 i 条带上的左边第一个符号，S_0 是多带图灵机的初始状态，然后依次执行原来为各带规定的操

作. 如果操作中包括移动，则把读写头原来所指的符号改为 a 类符号，移动后所指的符号改为 a' 类符号. 此外，在头 $n-1$ 条带上操作时，都不改变状态，只有完成第 n 条带的操作时才把状态变为 $(b'_{2j_2},\cdots,b'_{nj_n},S_1)$，其中诸 b'_{ij}是 n 个读写头现在所注视的符号，S_1 是原来多带图灵机进入的新状态.

读者也许要问：每当完成第 i 条带的操作后，如何找到第 $i+1$ 条带的操作位置呢？这就要靠 a' 类符号来指示位置了. 我们可以在第 $i+1$ 条带上先往右找，如果找到数据的尽头还没有，就再往左找，由于每条带上 a' 类符号是存在且唯一的，我们总能找到它作为操作的位置.

细心的读者还会问，在带上找 a' 类符号时，怎样才能发现"数据的尽头"呢？因为遇到空白符号不等于说前面就没有数据了，这样，在一条无穷长的带上可能会无休止地找下去. 为了解决这个问题，可以在开始计算时在每条带数据的右端设一个右界符号. 每当执行读写操作后读写头右移到一个右界符号上时，就把它往右推一格，这样，向右搜索 a' 类符号时，一旦遇到右界符就知道所找的 a' 类符号在左边，应该回头了.

第三个变种也许读者自己也能想到，是多带多道图灵机. 我们照样有：

定理 2.3 多带多道图灵机和标准图灵机有相同的计算能力.

读者不妨试一下自己证明它.

前面讲的图灵机的三个变种都是在结构上把图灵机复杂化. 我们已经看到，这种复杂化并不能提高图灵机的计算能力. 现在我们考察另一个方向的变种，即图灵机结构的简单化，看看这种简单化会不会降低图灵机的计算能力.

为此，我们考察图灵机的第四个变种：单向无穷图灵机. 这

种图灵机的带子在一个方向无限延伸，而在另一方向有尽头(好像一条半直线). 可以证明下列定理：

定理 2.4 单向无穷图灵机和标准(即双向无穷)图灵机有相同的计算能力.

证明的思想是这样的：假设图灵机的带子只向右方无限延伸，我们从带子的最左一格开始，给全部格子按自然数序列标上号，并且规定：奇数号的格子对应于双向无穷图灵机的带上从某个初始格子开始的向右部份，而偶数号的格子对应于双向无穷图灵机带上初始格子的左边部份. 无疑，这两种图灵机带上的格子间能够建立一一对应关系. 同时，对应于双向无穷图灵机的每个状态 S，在单向无穷图灵机中有两个状态 S 和 S' 与之对应. 其中 S 表示状态内容为 S 且读写头目前位于奇数格上，S' 表示状态内容为 S 且读写头目前位于偶数格上. 凡处于 S 状态时，移动读写头均跳过偶数格(例如可以通过令奇数格和偶数格有不同的符号来实现)，而处于 S' 状态时，移动读写头均跳过奇数格. 当读写头通过带子最左一格而“转弯”时，S 和 S' 状态交换. 具体的证明过程读者不妨自行补出.

下面的两个变种是香农给出的.

定理 2.5 只有两个符号的图灵机和标准(任意有限多个符号的)图灵机有相同的计算能力.

定理 2.6 只有两个状态的图灵机和标准(任意有限多个状态的)图灵机有相同的计算能力.

图灵机的信息存储能力取决于它的状态数和符号数. 因此，用表面上比较简单的图灵机去模拟表面上比较复杂的图灵机时，一般总是用增加状态数和/或符号数的办法来解决. 而状态数和符号数之间又有某种互补性. 定理 2.5 限制图灵机的符号数，其结果必然是增加它的状态数. 反之，定理 2.6 限制图灵机

的状态数，其结果必然是增加它的符号数．香农证明，具有 m 种符号和 n 种状态的图灵机可用只有两种符号，但有 $8mn$ 种状态的图灵机模拟，也可用具有 $4mn+m$ 种符号，而只有两种状态的图灵机来模拟．这是定理 2.5 和定理 2.6 的量化内容．

第三章　递归函数

——以有穷构造无穷的必由之路

让我们从讲一个故事开始.

一个没有去过北京的人问:天安门是什么样子?去过北京的人答道:天安门有个城楼,城楼上有个国徽,国徽里有个天安门,天安门有个城楼,城楼上有个国徽,国徽里有个天安门,……

读者不难看出,这个对天安门的介绍可以无穷无尽地继续下去.因为,为了要说清北京天安门的样子,必须先说清天安门城楼上国徽里天安门的样子.为了要说清国徽里天安门的样子,又必须先说清国徽里天安门城楼上国徽里天安门的样子,如此等等.这条天安门——城楼——国徽——天安门的链是无穷无尽的.总之,为了要说清天安门,必须先知道天安门,用天安门来解释天安门.在数学上,这种现象就叫做递归.递归是一个十分重要的概念.在数学里,有关计算的许多概念都要涉及递归.

数学上最简单的用递归方法来计算函数值的例子也许是阶乘了.自然数 n 的阶乘 $n!$ 定义为 $n!=n\cdot(n-1)\cdot(n-2)\cdot\cdots\cdot 2\cdot 1$.如果用 $f(n)$ 来表示求 $n!$ 的函数,则 $f(n)$ 可以写为:

$$f(0)=1$$
$$f(n)=n\cdot f(n-1) \qquad n>0 \tag{3.1}$$

这表示：为了要知道 $f(n)$ 的值，一般地说，需先知道 $f(n-1)$ 的值. 为了要知道 $f(n-1)$ 的值，一般地说，需先知道 $f(n-2)$ 的值，……，一直到 $f(0)$ 的值，即 1. 然后再一步步地按上边的公式倒推过来最终算出 $f(n)$ 的值. 这里的"一般地说"相当于"当 f 的变元大于 0 时".

刚才的计算方法可以说是"倒退着"做的，从 n 退到 $n-1$，从 $n-1$ 退到 $n-2$，……. 如果一开始就把计算的方向反过来：先求出 $f(0)$ 的值，再由 $f(0)$ 的值求出 $f(1)$ 的值，由 $f(1)$ 的值求出 $f(2)$ 的值，……，这在数学上叫递推.

递推的例子有很多，例如，著名的菲波那契数就可以用下列递推公式计算：

$$F(0)=1, \qquad F(1)=1,$$
$$F(n)=F(n-1)+F(n-2), \quad n>1. \tag{3.2}$$

由此公式，我们算出：

$$F(2)=2, \quad F(3)=3, \quad F(4)=5,$$
$$F(5)=8, \quad F(6)=13, \cdots\cdots$$

据说，菲波那契公式是在推算兔子繁衍规律时得到的.

表面上看来，似乎递归和递推差不多，实际上是不一样的. 并非所有的递归都能表示成递推. 例如本节一开始讲的天安门的例子，若表示成数学形式便不能化成递推. 因为这种递归是没有解的. 但是，有解的递归也不一定都能直接表示成递推的形式，例如

$$f(n)=\frac{1}{2}\cdot f(n-1) \tag{3.3}$$

有一个解 $f(n)\equiv 0$，但这个公式并未给出递归的终点（亦即递推的起点），所以不能看成是递推公式.

在数学上，递归是一个非常重要、非常基本的概念，它不但定义了一种基本的计算原则，而且给为什么是可计算的划定了一个界线. 递归函数理论断言说：一切可计算的函数都可以归结为几种基本的递归函数的组合. 在这里，首先要知道的是所谓原始递归函数. 原始递归函数共有三个，我们暂且认定它定义在整数集上，以后我们还要扩展它的定义域.

1. 后继函数： $S(x)=x+1$.

2. 零函数： $N(x)\equiv 0$. (3.4)

3. 单位函数： $U_i(x_1,\cdots,x_n)=x_i$.

在这三种原始递归函数之上定义了两类基本操作. 第一类叫函数复合，就是把一个函数放到另一个函数的参变量的位置上. 例如，我们可以推得：

$$S(S(S(x)))=x+3,$$

$$U_2(N(x),S(N(x)))\equiv 1,$$

等等. 一般的写法是：如果已知 $f(x_1,\cdots,x_n)$ 是原始递归函数，$g_1,\cdots,g_n$ 也是原始递归函数（为清晰起见，g_i 的诸参数皆省去），则 $f(g_1,\cdots,g_n)$ 也是一个原始递归函数，可以把它写成 $h(y_1,\cdots,y_m)$ 的形式，这里诸 y_i 构成的集合即是 $g_1,\cdots,g_n$ 中所有参变量的并集.

在原始递归函数上定义的第二类操作叫原始递归. 就是用类似于前面所说的递推的办法来定义新的原始递归函数. 我们以前面讲过的阶乘函数为例子，看看它是如何通过原始递归的方式定义的，把阶乘函数改写为如下形式：

$$\begin{aligned} &f(0)=1, \\ &f(n+1)=\mathrm{mul}(S(n),f(n)), \qquad n\geqslant 0 \end{aligned} \tag{3.5}$$

在这里，$S(n)$ 是前面已定义过的一个原始递归函数. mul 是一个新函数，表示乘法. $\mathrm{mul}(x,y)=x\cdot y$. 我们只要证明 mul 是原始

递归函数，那末，上面的式子就可看成是以已知的原始递归函数按递推方式定义新的原始递归函数的一种表示方法. mul 可以写成如下形式：

$$\begin{aligned} &\mathrm{mul}(0,y)=0, \\ &\mathrm{mul}(x+1,y)=\mathrm{add}(\mathrm{mul}(x,y),y). \end{aligned} \tag{3.6}$$

这里 add 是加法函数，$\mathrm{add}(x,y)=x+y$. 于是，问题又归结为考察是否可以用已知的原始递归函数以递推方式定义函数 add. 为此，我们把 add 写成如下形式：

$$\begin{aligned} &\mathrm{add}(0,y)=y, \\ &\mathrm{add}(x+1,y)=S(\mathrm{add}(x,y)). \end{aligned} \tag{3.7}$$

S 是已知的原始递归函数，这里再没有新的函数，问题解决了. 至此，我们用例子解释了什么是原始递归操作. 现在用形式化的方法来总结一下. 一个函数 $h(x_1,\cdots,x_n)$ 称为是可以用原始递归方式定义的，如果存在已知的原始递归函数 $f(x_2,\cdots,x_n)$ 和 $g(x,y,x_2,\cdots,x_n)$，使得下列两式成立：

$$\begin{aligned} &h(0,x_2,\cdots,x_n)=f(x_2,\cdots,x_n), \\ &h(x_1+1,x_2,\cdots,x_n)=g(x_1,h(x_1,\cdots,x_n),x_2,\cdots,x_n). \end{aligned} \tag{3.8}$$

作为一个练习，我们邀请读者把上面的阶乘，乘法，加法三个函数再严格地用这个形式写一遍. 请注意，用 S，N 和 U 三个基本的原始递归函数为出发点，反复施行各种可能的复合操作和原始递归操作，其结果就是全体原始递归函数类.

原始递归函数类的构造方法虽然简单，却不能小看了它. 人们在日常计算中用到的函数，几乎都可以归入这个类. 那末，有没有不是原始递归函数，但却可以用递归方法计算的函数呢？有的，这就是由希尔伯特的学生阿克曼构造的函数，人称阿克曼(Ackermann)函数，该函数的定义如下：

$$A(0,n)=n+1,$$
$$A(m,0)=A(m-1,1), \tag{3.9}$$
$$A(m,n)=A(m-1,A(m,n-1)).$$
这个函数的特点是有两重递归，因此它的值增长非常之快. 让我们来计算几个值.

$$A(1,n)=A(0,A(1,n-1))=A(1,n-1)+1,$$

由此可直接得到：

$$A(1,n)=A(1,0)+n=A(0,1)+n$$
$$=n+2.$$

阿克曼函数的两个变元不是对称的，第一个变元比起第二个变元来，对函数值的增长起的作用更大. 为此我们只需看一下它的开头几个值：

$$A(2,n)=2n+3,$$
$$A(3,n)=2^{n+3}-3, \tag{3.10}$$
$$A(4,n)=2^{2^{\cdot^{\cdot^{2}}}}\}n+3-3.$$

例如，$A(4,1)=2^{65536}-3$. 有兴趣的读者可以自己推导一下上面这几个式子.

定理 3.1 阿克曼函数不是原始递归函数.

证：作为第一步，先证明阿克曼函数对两个变元都是单调递增的. 这一步又分成如下几个小步：

1. 证明 $A(m,n)>n$. (3.11)

对 m 用归纳法. 当 $m=0$ 时此结论成立，因为根据定义有

$$A(0,n)=n+1>n.$$

现设此结论对某个 m 成立，我们对 n 施行归纳法. 当 $n=0$ 时结论为真，因为由定义及归纳假设知

$$A(m+1,0)=A(m,1)>1>0.$$

现设此结论对某个 n 成立，则

$$A(m+1,n+1)=A(m,A(m+1,n))$$
$$>A(m+1,n)>n,$$

因此得到 $A(m+1,n+1)>n+1$.

2. 证明 $A(m,n+1)>A(m,n)$. (3.12)

对 m 用归纳法. 当 $m=0$ 时结论成立,因为

$$A(0,n+1)=n+2>n+1=A(0,n).$$

现设此结论对某个 m 成立,利用定义及上面的结果有

$$A(m+1,n+1)=A(m,A(m+1,n))$$
$$>A(m+1,n).$$

归纳法得证.

3. 由此立得:对 $n_1>n_2$,有

$$A(m,n_1)>A(m,n_2). \quad (3.13)$$

4. 证明

$$A(m+1,n)\geqslant A(m,n+1). \quad (3.14)$$

对 n 用归纳法,当 $n=0$ 时结论为真,因为

$$A(m+1,0)=A(m,1).$$

现设此结论对某个 n 为真,利用定义及上面的结果有

$$A(m+1,n+1)=A(m,A(m+1,n))$$
$$\geqslant A(m,A(m,n+1))\geqslant A(m,n+2).$$

归纳得证.

5. 证明

$$A(m,n)>m. \quad (3.15)$$

反复利用刚才的结果,可以证明

$$A(m,n)\geqslant A(0,m+n)=m+n+1>m.$$

6. 证明对 $m_1>m_2$,有

$$A(m_1,n)>A(m_2,n). \quad (3.16)$$

设 $m_1-m_2=k>0$,重复 k 次利用上面的结论得

$$A(m_1,n)\geqslant A(m_2,n+k)>A(m_2,n),$$

至此,阿克曼函数对两个变元的单调递增性已全部得到证明.

现在做第二步.

要证明对于任意的原始递归函数 $f(x_1,x_2,\cdots,x_n)$,有一仅依赖于 f 的常数 M,使对所有的 $x_1,x_2,\cdots,x_n$ 有

$$f(x_1,x_2,\cdots,x_n)<A(M,\max(x_1,x_2,\cdots,x_n)). \qquad (3.17)$$

1. 首先,证明它对三个基本的原始递归函数成立. 我们有:

$$S(x)=x+1<x+2=A(1,x),$$

$$N(x)=0<x+2=A(1,x),$$

$$U_i(x_1,\cdots,x_n)=x_i<x_i+2=A(1,x_i)$$

$$\leqslant A(1,\max(x_1,\cdots,x_n)).$$

最后一步是根据第二个变元的单调性.

2. 其次,证明它若对于某几个原始递归函数成立的话,则对于通过在这几个函数上实行两类基本操作后所得的原始递归函数也成立.

第一类基本操作是函数复合,设有

$$h=f(g_1,\cdots,g_m),$$

其中 f 和诸 g_i 都是原始递归函数,无妨假设诸 g_i 有相同的变元 $x_1,x_2,\cdots,x_n$. 根据归纳假设,存在 m 个常数 M_i,使

$$g_i(x_1,\cdots,x_n)<A(M_i,\max(x_1,\cdots,x_n))$$

对所有 i 成立,这可以改写为:

$$g_i(x_1,\cdots,x_n)<A(M,\max(x_1,\cdots,x_n)),$$

其中 $M=\max(M_1,\cdots,M_m)$.

于是,对 f 可作下列推导:

$$f(g_1(x_1,\cdots,x_n),\cdots,g_m(x_1,\cdots,x_n))$$

$$<A(M_0,\max(g_1,\cdots,g_m))$$

$$<A(M_0, A(M, \max(x_1, \cdots, x_n)))$$
$$<A(M', A(M'+1, \max(x_1, \cdots, x_n)))$$
$$=A(M'+1, \max(x_1, \cdots, x_n)+1)$$
$$\leqslant A(M'+2, \max(x_1, \cdots, x_n))$$

其中 $M'=\max(M_0, M)$. 上面每一步推导都可以在前面已证的结论中找到根据.

第二类基本操作是原始递归. 设有

$$h(0, x_2, \cdots, x_n)=f(x_2, \cdots, x_n),$$
$$h(x_1+1, x_2, \cdots, x_n)=g(x_1, h(x_1, \cdots, x_n), x_2, \cdots, x_n),$$

其中 f 和 g 都是原始递归函数. 且有

$$f(x_2, \cdots, x_n)<A(M_1, \max(x_2, \cdots, x_n)),$$
$$g(y_1, y_2, x_2, \cdots, x_n)<A(M_2, \max(y_1, y_2, x_2, \cdots, x_n)).$$

令 $M_3=\max(M_1, M_2)+3$, 然后分几个小步来证明, 其中以 $\bar{x}$ 代 $\max(x_1, \cdots, x_n)$.

1. 证明 $h(x_1, \cdots, x_n)<A(M_3-2, \bar{x}+x_1)$.

对 x_1 施行归纳法, 当 $x_1=0$ 时结论为真. 因为

$$h(0, x_2, \cdots, x_n)<A(M_1, \bar{x})<A(M_3-2, \bar{x}).$$

设此结论对某个 x_1 为真, 则它对 x_1+1 也为真, 因为

$$h(x_1+1, x_2, \cdots, x_n)=g(x_1, h(x_1, \cdots, x_n), x_2, \cdots, x_n)$$
$$<A(M_2, \max(x_1, h(x_1, \cdots, x_n), \bar{x}))$$
$$<A(M_3-3, A(M_3-2, \bar{x}+x_1))$$
$$=A(M_3-2, \bar{x}+x_1+1).$$

因此, 根据式(3.18)可以推得:

$$h(x_1, \cdots, x_n)<A(M_3-2, \bar{x}+x_1)$$
$$\leqslant A(M_3-2, \quad 2\cdot\max(\bar{x}, x_1))$$
$$<A(M_3, \max(\bar{x}, x_1))=A(M_3, \max(x_1, \cdots, x_n))$$

现在我们补证(3.18)，对 n 实行归纳.

当 $n=0$ 时断言成立，因为 $A(m+2,0)>A(m,0)$.

现设(3.18)对某个 n 成立：

$$A(m+2,n)>A(m,2n), \tag{3.18}$$

根据定义及归纳假设

$$\begin{aligned}A(m+2,n+1)&=A(m+1,A(m+2,n))\\&>A(m+1,A(m,2n))\geqslant A(m,A(m,2n)+1)\\&\geqslant A(m,2n+2).\end{aligned}$$

现在做第三步.要证明阿克曼函数不具备上面讲的原始递归函数的那种特殊性质，用反证法.如果阿克曼函数是原始递归函数，则必有常数 M，使

$$A(x,y)<A(M,\max(x,y)).$$

特别地，我们可以令 $x=y=M$，于是得到

$$A(M,M)<A(M,M).$$

这是一个矛盾的式子.这表明，阿克曼函数不可能是原始递归函数.证毕.

尽管阿克曼函数的值随变元值(尤其是第一个变元值)的上升而增长很快，但它毕竟是有简明的递推公式可以遵循的能行可计算函数.上面的定理表明，阿克曼函数不属于原始递归函数类.就是说，原始递归函数类不能概括所有的能行可计算函数，为此，必须对原始递归函数类加以扩充.

人们找到了扩充原始递归函数类的途径，它借助于一种新的操作，叫 μ 操作，定义如下：

设 $f(y,x_1,\cdots,x_n)$ 是一个函数，式子

$$h(x_1,\cdots,x_n)=\mu y[f(y,x_1,\cdots,x_n)] \tag{3.19}$$

定义了一个新的函数 h，对每一组具体的变元值 $(a_1,\cdots,a_n)$，h 的函数值是使 $f(y,a_1,\cdots,a_n)=0$ 的最小的 y 值.如果这样的值

不存在，则令 h 在$(a_1,\cdots,a_n)$处无定义，由于这个含义，μ 操作也称为极小化操作，在式(3.19)中出现的 μ 算子称为极小化算子.

例如，令 $f(y,x)=x-y^2$，则通过式(3.19)定义的函数$h(x)$有如下性质：它在 $x=n^2$ 处$(n=0,1,2,\cdots)$之值为 n，而在其余地方没有定义.

经如此扩充以后的函数类称为 μ 递归函数类，又称部分递归函数类(因为它不一定处处有定义). 如果免去细节，μ 递归函数可以定义如下：

1. 一切原始递归函数都是 μ 递归函数.

2. 对 μ 递归函数实施函数复合操作，原始递归操作和极小化操作(μ 操作)后得到的仍是 μ 递归函数.

处处有定义的 μ 递归函数(部分递归函数)称为全递归函数. 可以证明如下定理：

定理 3.2 阿克曼函数是全递归函数.

那末，μ 递归函数和图灵机的关系如何呢？可以证明，μ 递归函数恰好就是图灵机可计算的函数. 这可以分成如下两个定理：

定理 3.3 μ 递归函数一定是图灵机可计算的函数.

证明大意如下：首先证明三个基本的原始递归函数(S、N 和 U_i)是图灵机可计算的. 这比较简单. 其次证明：把函数复合操作，原始递归操作和极小化操作施行于图灵机可计算的函数，所得到的新函数仍然是图灵机可计算的. 与这三种操作相应的证明是类似的，我们举其中的原始递归操作为例予以说明. 假设函数 h 通过式(3.8)定义，并已知 f 和 g 都是图灵机可计算的，则计算 $h(x_1+1,x_2,\cdots,x_n)$的图灵机可设计如下：用$\bar{x}$表示数 x 在图灵机上的表示(参见前章的例子)，用$\overline{x_{2-n}}$表示图灵机上的

数据 $\bar{x}_2 0 \bar{x}_3 0 \cdots 0 \bar{x}_n$，其中 0 是图灵机上的空白符号. 该图灵机的算法是：对初始数据

$$\overline{x_1+1} 0 \bar{x}_{2-n}$$

进行加工：复制 x_1+1 份 $\bar{x}_{2-n}$，与原来的 $\bar{x}_{2-n}$ 连在一起，并把 $\overline{x_1+1}$ 改造为数据 $\bar{x}_1 0 \overline{x_1-1} 0 \cdots 0 \bar{1} 0 \bar{0}$，放在所有 $\bar{x}_{2-n}$ 的左面. 形成如下局面：

$$\bar{x}_1 0 \overline{x_1-1} 0 \cdots 0 \bar{1} 0 \bar{0} 0 \underbrace{\bar{x}_{2-n} 0 \cdots \bar{x}_{2-n}}_{x_1+2\text{ 份}} 0.$$

用最左面的 $\bar{x}_{2-n}$ 计算 $f(x_2,\cdots,x_n)$，它等于 $h(0,x_2,\cdots,x_n)$，用 $\bar{h}(0,x_2,\cdots,x_n)$ 取代这个 $\bar{x}_{2-n}$ 在图灵机上的位置，得到

$$\bar{x}_1 0 \overline{x_1-1} 0 \cdots 0 \bar{1} 0 \bar{0} 0 \overline{h(0,x_2,\cdots x_n)} 0 \underbrace{\bar{x}_{2-n} 0 \cdots 0 \bar{x}_{2-n}}_{x_1+1\text{ 份}}.$$

用 $\bar{0}$，$\overline{h(0,x_2,\cdots,x_n)}$ 及最左面的 $\bar{x}_{2-n}$ 计算 $g(0,h(0,x_2,\cdots,x_n),x_2,\cdots,x_n)$，得到 $h(1,x_2,\cdots,x_n)$，又用 $\overline{h(1,x_2,\cdots,x_n)}$ 取代上述三个数据在图灵机上的位置，得到

$$\bar{x}_1 0 \overline{x_1-1} 0 \cdots 0 \bar{1} 0 \overline{h(1,x_2,\cdots,x_n)} 0 \underbrace{\bar{x}_{2-n} 0 \cdots 0 \bar{x}_{2-n}}_{x_1\text{ 份}}.$$

如此反复下去，最终即可求出 $h(x_1+1,x_2,x_3,\cdots,x_n)$. 这里假定 f 和 g 都是全函数.

对部分函数证法类似.

定理 3.4 图灵机可计算的函数一定是 μ 递归函数.

这个定理证明比较复杂，要使用一些特殊的技术，故不再列出.

第四章 λ 演 算

——这才是严格的函数运算

函数的概念是大家很熟悉的，可是在熟悉函数概念的人中间，大概有相当一部份并未注意到传统的函数表示法的缺陷.为了说明这种缺陷，让我们来看一些例子.在式子

$$y=x^2+2x+1 \tag{4.1}$$

中，可以把 y 看作是 x 的函数，但 y 真的是 x 的函数吗？这取决于该函数原来的定义，这种定义不一定是唯一的.例如

$$\begin{aligned} f(x)&=x^2+2x+1, \\ g(x)&=x^2, \end{aligned} \tag{4.2}$$

就是两种可能的定义，于是，有可能是 $y=f(x)$，也有可能是 $y=g(x+1)$.在后一种情况下，y 实际上是 $x+1$ 的函数.为什么会出现这种混淆呢？首先是因为传统的函数表示法或者不显式指出函数的自变量(如式子(4.1))，或者把自变量的说明和函数体分开(如式子(4.2))，其次是因为传统的函数表示法对函数的定义和函数的应用不加严格区分.例如 x^2+2x+1 既可以看成是函数 $f(x)$的定义，又可以看成是函数 $g(x)$对变量值 $x+1$ 的应用(以 $x+1$ 代入自变量 x 中)，实际上，定义和应用在概念上

是有很大区别的.

问题还不止于此. 在多个变量的情况下，我们会遇到更多的麻烦. 例如

$$f(z,y,x)=x+y^2+z^3, \tag{4.3}$$

是一个三变量函数.若把它应用于值三元组(1,2,3),请问这三个值分别对应于哪三个自变量呢？我们在这里有意地颠倒等式两边变量名出现的次序,为的是防止有的读者误认为把三个值顺序代入三个变量就行了.当然你可以像许多人所做的那样,用语言来表达你想要的对应方式,如:"x 用 1 代入,y 用 3 代入,z 用 2 代入"之类.但这是语言,是非形式的,它不是公式,不是符号,不能用来作数学推演.

上面的函数还有另一个问题,如果我们把它改写为:

$$u=x+y^2+z^3, \tag{4.4}$$

则也可以把 x 和 y 看作参变量,而仅把 z 看作自变量.这样,对于 x 和 y 的每一对具体值(例如 $x=1,y=2$),可以相应地得到一个 z 的函数.于是,我们可以把 u 看成是这样的函数:它的定义域为整数对,而它的值域是 z 的函数.这就跟一般的函数很不一样了.更有甚者,我们还可以令 x 和 y 是定义在 z 变量上的函数(例如 $x=z^2,y=2z$),这样,u 又成了以 z 的函数对为定义域,又以 z 的函数为值域的一个函数.这样的函数在数学上称为高阶函数,或称泛函.当然,我们完全可以换一种观点,例如把 z 看作参变量,而把 x 和 y 看作自变量,于是又得到对 u 的一种新的函数解释.这种在函数结构层次上的变化,用传统的函数表示方法是难以写清楚的.

为了解决这些问题,以丘奇为代表的一些数学家研究了函数的 λ 表示法,以及建立在 λ 表示法基础之上的 λ 演算.这种表示法能统一处理各种值域,对函数和其它类型的值一视同仁,所

以在 λ 表示法中不再区分函数和泛函这两个概念. 它的基本形式是：

$$\lambda\langle\text{变元}\rangle.\langle\text{表达式}\rangle \tag{4.5}$$

用这种方法表示的函数叫 λ 表达式. 如

$$\lambda x.\ x^2+1 \tag{4.6}$$

就是一个 λ 表达式，其中显式地指出了 x 是变元. 把 λ 表达式应用于具体的变元值时(按通常的说法:把具体的值代入函数变元时)，用一对括号把 λ 表达式括起来，而把变元值置于括号对之后. 例如若取变元 x 的值为 1，则按上述规则可以写为：

$$(\lambda x.\ x^2+1)1.$$

其计算方法是把变元值代入表达式中的变元，去掉前面的 λ⟨变元⟩，并按通常方式计算此表达式. 在上例中：

$$(\lambda x.\ x^2+1)1=1^2+1=2.$$

变元可以有多个，此时应按某种确定的顺序排列，例如：

$$\lambda x.\ \lambda y.\ x^3+y^2 \tag{4.7}$$

就是两个变元的 λ 表达式，把它作用于变元值 $x=1$，$y=2$ 时，得

$$((\lambda x.\ \lambda y.\ x^3+y^2)1)2$$
$$=(\lambda y.\ 1+y^2)2=5.$$

注意计算的顺序是固定的，采取由里层到外层的一层层"归约"的方法，决不会发生前面所说的变元值与变元的对应含混不清的问题. 此外还可以看出 λ 表示法的一个优点：函数的值可以又是一个函数，而且是显式地表示出来的，因为如果我们只把前面的 λ 表达式应用于变元值 1，则

$$(\lambda x.\ \lambda y.\ x^3+y^2)1=\lambda y.\ 1+y^2,$$

这是通常意义上 y 的一个函数，这说明，用 λ 表示法可以体现函数的层次关系.

改变变元的次序会影响函数应用的值，如

$$(\lambda y.\lambda x.x^3+y^2)1)2=(\lambda x.x^3+1)2=9,$$

在注意到这一点的前提下，可以在多变元 λ 表达式中只留一个 λ，而把(4.7)式写成

$$\lambda xy.x^3+y^2. \tag{4.8}$$

注意这只是一种简写，含意未变，它的内层依旧是 $\lambda y.x^3+y^2$. 在这个子 λ 表达式中，变元 x 未在 λ 部分出现，它有什么意义呢？它叫做该子 λ 表达式的自由变元. 而在 λ 部分出现的变元，如 y，则叫做约束变元. 注意自由和约束是相对的，x 对于式(4.8)的 λ 表达式来说，依然是约束变元.

自由变元和约束变元的相对性还有另一层含义. 在另一个 λ 表达式

$$\lambda x.x^3+y^2 \tag{4.9}$$

中，x 和 y 的身份交换了一下. x 成了约束变元，而 y 成了自由变元，但函数体 x^3+y^2 并没有改变. 由此可见，不同的 λ 首部能对函数体中的变元赋予不同的含义. 因此，人们把这种在表达式前加上 λ 首部的操作，叫做 λ 抽象，是 λ 表示法中的一个基本概念.

必须说明，λ 抽象只能把表达式中的自由变元变成约束变元，而不能反过来，并且在同一个 λ 表达式中可能同一个变元的某些出现是被约束的，如 $x\lambda x.x$ 中最后的 x，而另一些出现是自由的，如其中的第一个 x.

下面，我们要进入形式化的讨论，首先，让我们来定义一个形式的 λ 演算系统. 一个这样的系统由下列五部分组成：

1. 由变元组成的无穷集合 V.

2. 由常量组成的集合 C，集合 V 和集合 C 中的元素统称为原子.

3. 由四个特殊符号：λ，·，(，) 组成的集合 S. 它们在形式

系统中通常称为组合符. 集合 V、C 和 S 两两不相交.

4. 以 V、C 和 S 三集合中元素为语法元的一组语法公式.

5. 一组转换规则

我们在前面已举过不少 λ 表达式的例子，这里就不再列出它的语法公式了，只是非形式地作如下的说明：不论是有 λ 首部或没有 λ 首部的表达式，一律称为 λ 表达式. 而作为一个形式化系统中的表达式，它不再含有前面用过的＋、－、× 等各种算术运算符. 因为在形式化系统中，一切未用形式方法说明其含义的符号都是禁止使用的. 在一个形式化的 λ 系统中，只有两种表达式，一种是原子，另一种是某个 λ 表达式对某个原子的应用. 至于原子，则除了前面提到的变元和常量是原子外，用一对括号括起的 λ 表达式也叫原子. 注意我们在说明 λ 表达式的语法时用了递归的方法.

现在举一些形式系统中 λ 表达式的例子：x，a，(y)，xy，$(abcxyz)$，$x(ab)(yz)c$，$\lambda x.\ y$，$\lambda x.\ x$，$\lambda x.\ \lambda x.\ xx$，$yx(\lambda x.\ z)$，$(\lambda x.\ y)(\lambda y.\ z)(\lambda z.\ x)$ 等等都是合乎定义的 λ 表达式. 读者可根据前面的定义自行验证. 注意我们一般用 x、y、z 等表示变元，而用 a、b、c 等表示常量. 不难看出，任何一个复杂的 λ 表达式，都是用三种基本的粘合剂：λ 抽象，应用和括号，粘合而成的，其中应用的次序取左结合，如 $abc=(a(b))c$ 而不等于 $a(b(c))$，结合律在此不成立.

为了说明转换规则，需先说明什么叫置换. 置换在 λ 演算中有着特别重要的意义. 它的定义是这样的：令 x 是变元，M 和 N 是表达式，则置换 $[N/x,M]$ 表示用 N 代替 x 在 M 中的所有自由出现. 置换所得的结果是：

1. M，若 x 不在 M 中自由出现.

2. N，若 $M=x$.

3. $[N/x, L][N/x, R]$，若 $M=LR$，这里 L 和 R 都是表达式.

4. $([N/x, P])$，若 $M=(P)$.

5. $\lambda y.[N/x, K]$，若 $M=\lambda y.K$，$x\neq y$，y 不在 N 中自由出现，K 是表达式.

6. $\lambda z.[N/x,[z/y,K]]$，若 $M=\lambda y.K$， $x\neq y$， y 在 N 中自由出现，$z\neq x$， z 不在 M 或 N 中出现.

在这六条规定中，只有第 5 和第 6 条需要一点解释. 在第 5 条中，如果 y 在 N 中自由出现，则将得到完全无意义的结果. 例如实行置换 $[y/x, \lambda y.x]$ 的结果将得到 $\lambda y.y$，这是恒等函数，因为对任何 a 均有

$$(\lambda y.y)a=a,$$

这不是置换原来所要达到的目的. 那么，如果 y 在 N 中自由出现该如何办呢？第 6 条规定是对这种情况的一个补充，它提出了一种"换名"的办法，因为在上面的例子中，y/x 中的 y 和 $\lambda y.x$ 中的 y 本来没有任何关系. 由于在 $\lambda y.x$ 的 x 中并无 y 出现，所以把它写成 $\lambda y.x$ 或 $\lambda z.x$ 是完全一样的，这里的 z 是随便换一个名字. 如果我们把它写成 $\lambda z.x$，那么直接进行置换就无问题了，结果为 $\lambda z.y$. 对于复杂一点的情况，例如置换 $[y/x, \lambda y.(xy)]$，则可以把待置换表达式中的 y 一齐换名，例如把 $\lambda y.(xy)$ 换成 $\lambda z.(xz)$，然后再置换成 $\lambda z.(yz)$.

下面，我们要进入 λ 演算的演算部份了. 它以转换规则的形式出现，而转换的基础又是上面所说的置换，这些转换都是可逆的.

第一种转换叫 α 转换，实际上是一种换名，我们刚才已用过了. 设 $\lambda x.M$ 是一个 λ 表达式，则 α 转换就是把这个 λ 表达式变为 $\lambda y.N$，其中 $y\neq x$，y 不在 M 中自由出现，并且 $N=[y/x,$

$M]$，即 N 通过把 M 中所有 x 的自由出现代之以 y 而得到. 注意，这里的"自由"是相对于 M 而言的，不是相对于 $\lambda x.M$ 而言的，今后，我们用 $A \xrightarrow[\alpha]{} B$ 表示由 λ 表达式 A 经过 α 转换得到 λ 表达式 B. 例如：

$$\lambda x.xy(\lambda x.x) \xrightarrow[\alpha]{} \lambda z.zy(\lambda x.x),$$

$$\lambda x.xy(\lambda x.x) \xrightarrow[\alpha]{} \lambda x.xy(\lambda z.z),$$

第一个例子以 $xy(\lambda x.x)$ 为 M，其中前面的 x 是自由出现，因而和 λ 首部的 x 一起换成了 z. 后面的 x 是约束出现，不换. 第二个例子以子 λ 表达式 $\lambda x.x$ 中的 x 为 M，它是自由的，因此换成了 z，外面的 x 与此无关，不换.

第二种转换叫 β 转换，实际上就是实施函数应用. 设 $\lambda x.M$ 和 N 为任意的 λ 表达式，则把 $\lambda x.M$ 应用于 N 称为 β 转换，形式为：

$$(\lambda x.M)N \xrightarrow[\beta]{} [N/x, M]. \tag{4.10}$$

下面是几个 β 转换的例子：

$$(\lambda x.xy)z \xrightarrow[\beta]{} zy,$$

$$(\lambda x.yz)x \xrightarrow[\beta]{} yz,$$

$$(\lambda x.(\lambda y.yx)z)t \xrightarrow[\beta]{} (\lambda y.yt)z \xrightarrow[\beta]{} zt.$$

有时要首先用 α 转换解决变量名冲突，然后再实施正常的 β 转换，例如：

$$(\lambda x.(\lambda y.yx)z)y \xrightarrow[\alpha]{} (\lambda x.(\lambda t.tx)z)y$$

$$\xrightarrow[\beta]{} (\lambda t.ty)z \xrightarrow[\beta]{} zy.$$

第三种转换叫 η 转换，也称为外延转换. 设 $\lambda x.Mx$ 是 λ 表达式，x 不在 M 中自由出现，则转换：

$$\lambda x.Mx \xrightarrow[\eta]{} M \tag{4.11}$$

称为 η 转换. 作 η 转换的理由是:对任何一个表达式 N,均有(因为 M 中没有 x 的自由出现):

$$(\lambda x.Mx)N \xrightarrow[\beta]{} MN. \tag{4.12}$$

就是说,由于 $\lambda x.Mx$ 和 M 应用于任意变元值均得相同结果,我们把 $\lambda x.Mx$ 和 M 在 η 转换的意义下看成是同一个 λ 表达式. 通常,一个函数可按内涵或外延方式定义. 函数的结构和特性决定了它的内涵,函数值和变元值的对应则是它的外延. 两个函数在什么情况下视为相同?是仅凭它们的处延,还是应根据它们的内涵来定? 这在不同的系统中有不同的规定. η 转换在 λ 演算中引进了一种特殊的外延性,为达到更一般的外延性,我们还需要新的转换.

第四种转换叫 ξ 转换,也称抽象转换. 设 M 和 N 为 λ 表达式,且有

$$M \xrightarrow[\alpha,\beta,\eta]{} N \tag{4.13}$$

成立,则下列形式的转换称为 ξ 转换

$$\lambda x.M \xrightarrow[\xi]{} \lambda x.N. \tag{4.14}$$

这里 $\xrightarrow[\alpha,\beta,\eta]{}$ 表示它的左边可通过适当选择 α、β,η 转换而变成右边.

第五种转换叫 ζ 转换,也叫广义外延转换. 设 M 和 N 是 λ 表达式,x 不在 M 和 N 中自由出现. 如果允许 ξ 转换,且有

$$Mx \xrightarrow[\alpha,\beta,\eta]{} Nx \tag{4.15}$$

成立,则有

$$M \xrightarrow[\zeta]{} N \tag{4.16}$$

它的意思是:即使两个 λ 表达式应用于相同的变元值不一定得到相同的结果,但在 α、β 和 η 转换的意义下是相同的结果,则可以认为此两个 λ 表达式的广义外延相同,因而在 ζ 转换下被认

为是同一个 λ 表达式.

以上讲的这些转换,并不是每个 λ 演算系统都全部允许的,也不是各种 λ 演算系统所允许的转换都已包括在内. 由于允许转换的种类多少不同,各系统的功能强弱也不同,得到的理论结果也不一样,因此,在进行讨论时,往往要说明是"在允许哪几种转换的前提下". 打个比方,在几何学中,允许直角座标变换的是欧氏几何,允许仿射变换的是仿射几何,允许射影变换的是射影几何,等等. 在这里,α 转换属各系统公有,毋需指出,其它均需指明. 如 $\beta\eta$ 演算表示允许 β 转换和 η 转换的 λ 演算,$\beta\xi$ 演算表示允许 β 转换和 ξ 转换的 λ 演算,如此等等. 允许所有转换的系统称为完全 λ 演算系统.

λ 演算具有模拟其它形式系统的能力,例如,任一正整数 n 可表为

$$\lambda xy.\underbrace{x(x(x\cdots(x}_{n\text{个}x}y)\cdots).$$

采取这种表示法的原因是:对任一原子 f,均有:

$$(\lambda xy.\underbrace{x(x(x\cdots(x}_{n\text{个}x}y)\cdots)f=\lambda y.\underbrace{f(f\cdots f}_{n\text{个}f}(y)\cdots),$$

即用函数的 n 重嵌套来表示正整数 n. 在这个基础上,λ 演算可用来模拟一整套递归函数理论. 可以证明,λ 演算具有和部份递归函数相同的计算能力.

第五章　命题演算和谓词演算

——思维演算的符号体系

某村农民王某被害，有四个嫌疑犯 A、B、C、D. 公安局的五个侦察员各自发表了见解，甲说 A、B 中至少有一人作案；乙说 B、C 中至少有一人作案；丙说 C、D 中至少有一人作案；丁说 A、C 中至少有一人与此案无关；戊说 B、D 中至少有一人与此案无关. 众说纷纭，使负责破案的刑警队长抓耳挠腮，不知从何下手.

其实，如果请一位数理逻辑学家来当顾问就好了，破案与逻辑推理有很大的关系. 人的思维推理非常复杂，就像上面提到的案子那样，明明破案的线索已经包含在五位侦察员的意见中，但由于综合这些意见的过程是一个复杂的逻辑推理过程，因此通常不易一眼看穿. 为了使这种逻辑推理过程变得简单一点，早在三百年前，一位著名的德国数学家莱布尼茨就提出了一种设想. 他认为，如果能够创造一套表达概念的符号语言，并且把人类的推理过程用某种公式来表示，那么就能够发明一种思维演算，把逻辑推理过程转化为计算过程，这样，解决人与人之间争论的困难就可以像做一道数学题那样给以解决. 莱布尼茨的这个思想是非常富有启发性的，他成了现代数理逻辑的最早思想先驱.

可惜的是，莱布尼茨未能实现他的理想，莱氏的同时代人也未能完成他的夙愿. 直到一百多年以后，英国数学家布尔以他的布尔代数提供了一套初步可用的思维演算工具，才使这方面的研究有了本质性的进展. 一个布尔代数 $\overline{B}$ 是一个至少有两个元素的集合，在此集合上定义了加法和乘法两种运算，两种运算都满足交换律($a+b=b+a$， $ab=ba$). 两种运算的组合满足分配律 ($a(b+c)=(ab)+(ac)$， $a+(bc)=(a+b)(a+c)$). 且 B 中有零元素 0 和幺元素 1(对任何 x 有 $x+0=x$， $x\cdot 1=x$)，并且对任何 $x\in\overline{B}$ 皆有 $x'\in\overline{B}$，使 $x+x'=1$， $xx'=0$.

最简单的布尔代数是 $\overline{B}=\{0,1\}$，只有两个元素，并且满足下列加法和乘法规则：

+	0	1
0	0	1
1	1	1

×	0	1
0	0	0
1	0	1

(5.1)

在这个最简单的布尔代数中，通常把 0 解释为假，把 1 解释为真(或反过来). 把 $a+b=0$ 解释为 a、b 都为假，把 $a+b=1$ 解释为 a、b 至少有一为真. 把 $ab=0$ 解释为 a、b 至少有一为假. 把 $a\cdot b=1$ 解释为 a、b 皆真.

让我们再回过头来看本节开头的例子. 我们不妨构造一个从嫌疑犯集合 $\{A,B,C,D\}$ 到布尔代数 $\overline{B}=\{0,1\}$ 的映射，并规定：把真正的罪犯都映射为 1，非罪犯映射为 0. 由于我们目前还不知道谁是罪犯，因此暂时仍以 A、B、C、D 表示它们的映象，于是我们得到如下方程

$$A+B=1, \tag{5.2}$$

$$B+C=1, \tag{5.3}$$

$$C+D=1, \tag{5.4}$$

$$A \cdot C=0, \tag{5.5}$$

$$B \cdot D=0. \tag{5.6}$$

用 C 乘式(5.2)两边，并注意到式(5.5)，可得 $BC=C$，再在新式子的两边加 B，并注意到式(5.3)，得到 $B(1+C)=1$. 由式(5.1)知必有 $B=1$. 又由式(5.6)知必有 $D=0$. 用同样的方法可得 $A=0$, $C=1$. 这说明 B、C 是凶手，而 A、D 为无辜，不难看出，思维推理在这里变成了公式演算.

读者会发现，上面的演算过程并没有完全形式化，机械化. 事实上，布尔也确实只是开了个头，他并没有完成数理逻辑(哪怕是数理逻辑的初级内容，即命题演算)的创建，这个任务留给了比布尔晚一辈的德国数学家弗雷格. 弗雷格不但完成了命题演算，构造了命题演算的第一个公理系统，而且还引进了量词，把命题演算发展成谓词演算. 在具体介绍命题演算和谓词演算之前，我们先要说一说数理逻辑发展的另一个推动力，它不是来自数学，而是来自哲学领域的研究和探索.

原来，数理逻辑的发展和分析哲学的兴起有很大的关系，分析哲学是半个多世纪以来在英美哲学中占主导地位的哲学思想，具有很大的影响. 分析哲学家认为，哲学的首要任务在于分析，他们不像过去的哲学家那样一上来就企图建立一个庞大的哲学体系，而是从具体的事物和概念开始，一点一点地进行分析. 他们认为，哲学家之间的意见不同和纷争，都是由于所使用的概念和语言的含混不清，只要能找到一种科学的形式分析或逻辑分析方法，并正确地使用它，哲学上的许多分歧就可以迎刃而解. 这种观点的核心部分是错误的. 因为哲学上有一些根本的分歧，如唯物论和唯心论之争，决不是由于概念和语言的混淆而引起的，企图用什么形式分析或逻辑分析方法来分析出唯物论

和唯心论“原来并无分歧”，或调和它们的对立，都注定是徒劳的．此外，分析哲学还有其它的一些错误，例如他们像许多唯心主义经验论者那样，也把人的经验看成是脱离客观的、纯主观性的东西，拒绝研究经验之外的客观实在等等，这些都说明分析哲学带有唯心主义倾向．但是，分析哲学家为了贯彻他们的主张而提出的许多方法和技术，却是值得我们借鉴和使用的．其中，数理逻辑便是早期分析哲学家发明和使用的主要工具之一．

刚才说的弗雷格由于基本上完成了命题演算和狭谓词演算而为分析哲学提供了有力的逻辑分析工具，他本人也被推崇为分析哲学的思想先驱．下面我们来介绍一下这两种演算的基本内容．

命题演算可以看作是前面所说的最简单布尔代数的扩充．扩充在下列几方面：第一，它不止含有两个元素（布尔代数的 1 和 0 在命题演算中表为真和假），在真（用 T 表示）和假（用 F 表示）之外增加了一批变量元素，这些变量元素的真假值随赋值而定．第二，它不止有两个操作＋和×，而是有 $\vee$（或），$\wedge$（与），$\sim$（非），$\rightarrow$（蕴含），$\equiv$（等价）等多项操作．其中“或操作”和“与操作”相当于布尔代数的“＋操作”和“×操作”．

命题用某种符号表示（如一个英文字母或一个英文字）．举例来说，用 A 表示命题“张三是罪犯”，B 表示命题“李四是罪犯”，则 $\sim A$ 表示命题“张三不是罪犯”；$A \vee B$ 表示命题“张三和李四两人中至少有一人是罪犯”；$A \wedge B$ 表示命题“张三和李四都是罪犯”；$A \rightarrow B$ 表示命题“若张三是罪犯，则李四也是罪犯”；$A \equiv B$ 表示命题“张三和李四要么都是罪犯，要么都不是罪犯”．这些命题中有几个是由两个子命题组合而成的．凡是由多个命题组合而成的命题称为复合命题．

确切地说，一个复合命题的真假值由下表给出．$\sim A$ 不算复

合命题，它的值也在表中给出.

A	B	$\sim A$	$A \vee B$	$A \wedge B$	$A \to B$	$A \equiv B$
T	T	F	T	T	T	T
T	F	F	T	F	F	F
F	T	T	T	F	T	F
F	F	T	F	F	T	T

这张表称为真值表，当需要分析一个复杂的命题时，常用这种真值表方法.建议读者仔细研究一下它，并多用一些实际例子给予验证.表中最值得注意的是 $A \to B$ 这一列.一般人对于前两行结论(即($T \to T$)之值为 T，($T \to F$)之值为 F)是可以接受的.对于后两行结论($F \to T$)之值为 T 和($F \to F$)之值为 T)则可能会有些怀疑.因为那个命题的内容是：如果 A 为真，则 B 为真.对于 A 为假的情况，它可是什么也没有说.那么这后两个结论是从哪里来的呢？实际上这些结论在古希腊的逻辑学家那里就已经有了.这些古代的先哲们研究过 $A \to B$ 形式的蕴含关系，并把它定义为 $\sim A \vee B$.读者不妨验证一下，$A \to B$ 和 $\sim A \vee B$ 的真值是完全一样的.从那以后，$A \to B$ 的这个定义就一直被延用下来.有什么道理没有？没有很多道理，只是因为人们感到这样的定义是严密的，并且用起来很方便.例如在著名数学家和逻辑学家罗素和怀特海的巨著《数学原理》中，就以这个定义(俗称实质蕴含)为主要工具来表达数学基础.当然它也并非完全没有问题，我们在下面还要提到.

到目前为止，我们对命题演算的介绍是非形式化的，现在需要把它作为一个形式系统而严格地引进.这样的形式系统不是唯一的，下面给出的是其中的一种.

一个形式的命题演算系统包括语法和语义两部份. 语法部份指明哪些符号以及符号的组合是允许的(合法的). 语义部份给出每个合法的符号组合(称为合式公式)与真值表的对应,以及合式公式之间的对应. 具体地说:

给定一组命题变量 $A,B,C,D,E,\cdots\cdots$;两个操作符$\sim$和$\vee$,以及括号符 (和),则

1. 命题变量都是合式公式.

2. 若 p 是合式公式,则$\sim p$ 也是合式公式.

3. 若 p、q 是合式公式,则$(p\vee q)$也是合式公式.

4. 若某符号组合 p 被定义为另一已知的合式公式 q,则 p 也是合式公式.

5. 除此之外没有其它合式公式.

在这里,"p 定义为 q"理解为 p 和 q 有相同的真值表.

该形式系统引进了如下几项定义:

1. $A\rightarrow B$ 定义为$\sim A\vee B$.

2. $A\wedge B$ 定义为$\sim(\sim A\vee\sim B)$. (5.7)

3. $A\equiv B$ 定义为$(A\rightarrow B)\wedge(B\rightarrow A)$.

4. 以任意的合式公式代入上述三条定义的两端,其中同一命题变量的所有出现用同一合式公式代入,则得到的仍旧是定义关系.

形式系统要有一组公理,公理是恒取真值的合式公式. 本系统公理共有四条:

1. $p\vee p\rightarrow p$.

2. $p\rightarrow p\vee q$.

3. $p\vee q\rightarrow q\vee p$. (5.8)

4. $(q\rightarrow r)\rightarrow(p\vee q\rightarrow p\vee r)$.

形式系统还需要一套推导规则,它们是:

1. 代入规则：合式公式中出现的命题变量可以用另一个合式公式代入. 如果原来的合式公式是恒真的，并且同一命题变量的所有出现均以同一合式公式代入，则所得的新合式公式仍然恒真.

2. 分离规则：若合式公式 p 和 $\sim p \vee q$ 皆取真值，则 q 也取真值.

3. 置换规则：若 p 被定义为 q，则任一合式公式 r 中的所有 p 可被同时置换为 q 或反之. 若 r 在置换前取真值，则置换后仍取真值.

4. 操作符的优先次序是(从小到大)：$\equiv$，$\rightarrow$，$\vee$，$\wedge$，$\sim$. 根据这个次序，在不改变真值的前提下，可以省去部份或全部括号对.

最后，形式系统为合式公式规定的真值表与我们在前面给出的一样. 从操作符 $\rightarrow$，$\wedge$，$\equiv$ 的定义可以看出，它们的含义不是独立的，可以用其它的操作符构造出来.

现在让我们用这个形式系统推导一下，看能推出什么东西来.

在第二条公理中用 $\sim q$ 代替 q，得到

$$p \rightarrow p \vee \sim q,$$

再运用公理 3 和定义 1，即得到：

$$p \rightarrow (q \rightarrow p). \tag{5.9}$$

这表示：若 p 为真，则任何命题 q 均可导出 p 为真. 现在我们再一次使用公理 2，但是用 $\sim p$ 代替 p，得到

$$\sim p \rightarrow \sim p \vee q,$$

再次运用定义 1，得

$$\sim p \rightarrow (p \rightarrow q). \tag{5.10}$$

这表示：若 p 为假，则从 p 可推知任意命题 q 为真. (5.9)和

(5.10)在数理逻辑上被称为蕴含悖论，引起过极大的争议.这在前面我们讨论真值表时已经提到过一点了.据说当罗素做演讲时，有的听众对此不服并问道：如果从一个假命题可以推出任意命题为真，那么从 2+2=5 能够推出你就是教皇吗？罗素答道，2+2=4，因此 4=5，两边减去 3，得到 1=2，我和教皇是两个人，1=2 说明两个人是一个人，所以我就是教皇.实际上，这种道理一般人也可以理解，中国人常说：若要某某情况发生，除非太阳从西边出来.由于“太阳从西边出来”是一个假命题，所以任何命题都可由此导出.

关于命题逻辑的介绍大致就到这里，我们为此花去了较多的篇幅，因为命题逻辑是基础，懂得了命题逻辑，谓词逻辑也就不难掌握了.什么叫谓词呢？就是可以带变元的命题(谓词是数理逻辑的术语，不是我们学英语或汉语语法时所用的谓语概念).在命题演算中，我们见到两类命题：命题常量(如 T、F)和命题变量.命题常量的真假值在任何情况下不会变，命题变量的真假值由赋值(或称指派)决定.在一个具体的命题公式(合式公式)中，可以指定某命题变量取真值或假值.而谓词则不然，它好象是一个可取真、假两种值的函数，在每个具体场合下到底取真值还是取假值取决于该谓词的各变元被什么具体值代入.例如我们有命题：雪是白的、煤球是黑的、雪是黑的、煤球是白的.显然在常识意义下前两个命题取真值，而后两个命题取假值.在谓词演算中，我们可以把白和黑看作两个谓词名，并引进如下两个带变元的谓词白(x)、黑(x).于是，很自然地我们有白(雪)和黑(煤球)取真值，而白(煤球)和黑(雪)取假值.

如果仅看这个例子，那么数理逻辑中的谓词真有点象英语语法中的谓语了.实际上，数理逻辑中的谓词可以有任意多个变元，一个谓词本质上表示一个(存在于它的诸变元之间的)关系，

而并不表示日常语言中的一个主宾式语句.这正是数理逻辑与传统的形式逻辑之间的一大区别.传统逻辑只研究“A 为 B”形式的判断.这种做法受到了著名逻辑学家德摩根的批评,他认为逻辑的研究范围不应限于主宾式语句,而应扩展到包括一般关系的语句,即“$x_1,x_2,\cdots,x_n$ 具有关系 R”形式的语句.这个主张被逻辑学界所接受,因而使数理逻辑中的谓词具有现在这样的形式.例如,一个人的档案数据(姓名、籍贯,性别,出生时间,父名、母名)可以构成一个关系 PD.如果把 PD 看作一个谓词名,则 PD(毛岸英,湘潭,男,1923,毛泽东,杨开慧)和 PD(周恩来,绍兴,男,1898,周贻能,万冬儿)都取真值.

除了引进变元和采取关系描述的形式以外,谓词演算的另一个特点是引进了两个量词:全称量词∀ 和存在量词∃ ,这基本上是弗雷格的功劳,但使它具有今天这样形式的是另一位逻辑学家皮尔斯.它们的含义是:

$\exists\ x_1,\cdots,x_n,p(x_1,\cdots,x_n)$,表示存在一组变元值$(x_1,\cdots,x_n)$,使 $p(x_1,\cdots,x_n)$之值为真.

$\forall\ x_1,\cdots,x_n,p(x_1,\cdots,x_n)$,表示对所有的变元值$(x_1,\cdots,x_n)$,$p(x_1,\cdots,x_n)$之值皆为真.

上面的说明还不是最一般的,它可以在如下几个方面拓广:

1.谓词 $p(x_1,\cdots,x_n)$可代之以任意的合式公式.

2.量词∃ 和∀ 后面的变元名集可以是后面合式公式中变元名集的子集.

3.量词∃ 和∀ 可以多重嵌套、交叉使用.

谓词演算中合式公式的构造方式和命题演算中的类似,只不过是以谓词取代命题,并加进两个量词.被量词约束的变元称为约束变元,否则称为自由变元.除此之外,在谓词演算中还允许引进函数符号,如 f,g,h,等等.函数符号后面也可以跟零个,

一个或多个变元.后跟零个变元的函数符号就是常量，这种函数称为零秩函数.函数符号(以及它的变元)出现在谓词变元的位置上，也可出现在另一函数符号的变元位置上，就是说，函数可以嵌套.例如

$$p(x,f(g(y,z),h(x)),h(g(x,f(z,y))))$$

是一个合法的谓词，其中 p 是谓词符号，f、g 和 h 都是函数符号. x、y、z 是变元.

现在看几个谓词合式公式的例子.

$$\exists x_1,\cdots,x_6,\mathrm{PD}(x_1,\cdots,x_6).$$

这个合式公式的语义可以解释为世界上至少有一个家庭.

$$\forall x(\mathrm{human}(x)\rightarrow \mathrm{PD}(x,\mathrm{bp}(x),\mathrm{sex}(x),\mathrm{bt}(x),\mathrm{father}(x),\mathrm{mother}(x))$$

可解释为每人都有一个家庭.其中 human 表示"是人"，bp 和 bt 分别表示籍贯和出生时间，注意 human 和 PD 是谓词名，其余都是函数名(此处名与符号是一个意思).为了显示函数嵌套，也可把 mother(x)改为 wife(father(x))，这样改的问题是：父亲的妻子可能不是母亲，与该谓词变元的原意相悖.

对于存在量词和全称量词混合使用的情况，也许用数学定理来做例子是最合适的了.比如，我们知道无穷级数

$$\sum_{n=1}^{\infty} n^{\frac{1}{1+\varepsilon}}$$

是收敛的，其中 ε 是任意正数，用 $S(\varepsilon,N)$来表示它的部分和

$$\sum_{n=1}^{N} n^{\frac{1}{1+\varepsilon}},$$

则我们不妨用如下的谓词公式来说明它的收敛性质：

$$\forall \varepsilon_1[\mathrm{gr}(\varepsilon_1,0)\rightarrow\forall \varepsilon_2[\mathrm{gr}(\varepsilon_2,0)\rightarrow\exists N[\mathrm{int}(N)$$

$$\wedge\,\mathrm{gr}(N,0)\wedge[\forall N_1,N_2,\mathrm{int}(N_1)\wedge\mathrm{gr}(N_1,N)$$

$$\wedge \mathrm{int}(N_2) \wedge \mathrm{gr}(N_2, N_1) \longrightarrow \mathrm{ls}(\mathrm{sub}(S(\varepsilon_1, N_2), S(\varepsilon_1, N_1)), \varepsilon_2)]]]].$$

翻译成语言就是：对任何 $\varepsilon_1>0$，$\varepsilon_2>0$，必存在正整数 N，使得对任意 $N_2>N_1>N$ 皆有 $S(\varepsilon_1,N_2)-S(\varepsilon_1,N_1)<\varepsilon_2$，当然这个式子还可化简，它的表示法也不是唯一的. 但至少我们可以看到谓词表达法的威力.

与命题演算一样，谓词演算也有一套演算规则，本章中介绍的谓词演算称为狭谓词演算，或一阶谓词演算. 狭谓词演算也有它的形式化系统，也有语法部分和语义部分. 也是首先给出一套符号，然后再规定符号的组合规则，形成合式公式，然后用已知的操作符定义新的操作符，然后给出一组其值恒真的公理，作为推导的出发点，然后再给出一组推导规则，等等. 其过程和给出命题演算的形式系统的过程差不多，但在技术细节方面要复杂得多，对此详细论述将超出这本小册子所允许的篇幅.

不能忘记说明：狭谓词演算最重要的性质之一是：它具有和递归函数（或图灵机）相同的计算能力. 也就是说，它能描述一切能行可计算的过程.

还有一点不能忘记说明：谓词演算是建立在形式系统上的一套演算，其中的一切符号本身没有任何意义. 所有的“意义”都是人为地赋予的. 例如我们在上面用 PD 表示人事档案，用 human 表示“是人”，用 bp 表示籍贯，用 father 表示父亲，等等，这些都是我们的主观解释，一个谓词演算系统决不会懂得这些解释，也不会从这些解释中主动推出任何结论. 例如，若以 teacher(x,y)表示 x 是 y 的老师，student(y,x)表示 y 是 x 的学生. 则形式系统决不会从 teacher（熊庆来，华罗庚）自动推出 student（华罗庚，熊庆来），除非你明确地把蕴涵公式

teacher(x, y)→student(y, x)写出来．同样，形式系统并不会自己知道白(雪)为真，而白(煤球)为假，除非你显式地写出这一点．

第六章　文法、语言和自动机

——三个等级森严的家族

在很多年以前，欧美殖民主义者曾经靠大批贩卖奴隶发了一大笔横财. 看过电影《海囚》的人都会记得这种悲惨的场面. 殖民地国家的劳动人民被当作牛马运到海外去开矿、修路、做苦工，过着暗无天日的生活. 从此，他们的子孙后代也就在这异国他乡繁衍开了. 由于这些人来自不同的国家，使用不同的语言，他们在共同生活中为了交流思想的需要，逐渐发展起了一种与每个人的祖国语言都不同的语言，称为克里奥尔语. 经语言学家研究：克里奥尔语不可能从这些人的祖国语言中的任何一种发展而来. 更有意思的是，各个不同地区独立形成的支流语言（我们统一称它们为克里奥尔语）居然有很多相似之处，并且它们之间的相似性要超过它们和其它自然语言（我们称某一个民族的人实际使用的语言为自然语言）之间的相似性. 这件事引起了语言学界的重视. 人们问道：既然克里奥尔语不是从任何一种语言发展而来，那么它是怎样诞生的呢？克里奥尔语的发现给一场带有哲学味道的争论添了一把火. 原来：对于儿童的语言学习机制，人们本来就有两种看法. 一种意见认为儿童没有任何先天的

语言学知识，他们的所有语言才能都是后天获得的.另一种意见则认为儿童先天头脑中就有一些语言模式，他们出生后在家庭和社会的影响下才固定了其中的一种模式，并在此基础上进一步发展其语言能力.著名语言学家乔姆斯基是后一种意见的代表.上述克里奥尔语的发现被持后一种观点的学者作为儿童具有先天语言才能的有力证据.他们还认为：在某种自然语言占主导地位的环境中，儿童的这种先天语言模式得不到自由发展的机会，它只能被环境溶化成当地的主导模式.只有在多种民族杂居的地区，在各种“洋泾浜”语滋生的环境中，儿童的先天语言模式才有成长发展的可能性.当然，以上观点只是学者的一种分析，还不能构成一种严格的论证.严格论证还得要把数学请出来（见第八章）.不过在此之前，我们先说一说语言学研究的发展情况.

据统计，世界上共有 5651 种不同的语言或方言.有不少古代语言已经死亡，数以千计的语言正在衰亡之中，使用它们的人数越来越少.经语言学家确认为独立的语言有 2790 种，其中 70%还没有相应的文字.经过分析研究的语言只有约 500 种，使用人数超过 6000 万的语言有 13 种，其中汉语位居榜首，在语言中使用汉语字符的人口占世界人口的 36%.

对这么复杂的语言体系进行研究是语言学家的光荣任务.随着整个自然科学和社会科学的发展，语言学的研究也走过了三个历史时期：历史比较语言学、结构主义语言学和转换生成语言学.历史比较语言学的基本思想和达尔文的生物进化论十分类似，它在对大量语言资料作分析比较的基础上，研究人类语言进化繁衍的历史.结构主义语言学则类似于生物解剖学，通过深入分析语言的结构和组成规则来搞清这个复杂体系的形成机制，特别是研究每一种语言区别于其它语言的特征.而转换生成

语言学却一反上面两个历史阶段的传统，它不把语言作为一个孤立的现象加以研究，而是把语言和使用语言的人结合起来进行研究. 特别是要搞清楚：人是怎么会使用语言的？父母和老师只给儿童传授过有限的语句集合，为什么儿童（特别是成长以后）会讲出几乎是无限多种不同的句子来？为了解释这个现象，乔姆斯基在 50 年代末、60 年代初创立了转换生成语言学. 乔氏理论的核心思想是：人的语言是由基本语法出发，经演绎过程而生成的，而这套基本语法基本上是生来就有的，只是在儿童成长过程中通过学习、调整成某种确定的模式. 由于从一个有限的语法出发，可以演绎出无穷多个句子来，因而人也有了说出无穷多种语句的能力.

为了介绍乔姆斯基的理论，我们要暂时离开自然语言的领域，只考虑形式语言. 形式语言是用数学方式定义的人工语言. 它的定义可以这样叙述：给定一组符号（一般是有限多个），称为字母表，以 Σ 表之. 又以 Σ^* 表示由 Σ 中字母组成的所有符号串（也称字，包括空字）的集合，则 Σ^* 的每个子集称为 Σ 上的一个语言. 例如，若令 Σ 为 26 个拉丁字母加上空格和标点符号，则每个英语句子都是 Σ^* 中的一个元素，所有合法的英语句子的集合是 Σ^* 的一个子集，它构成一个语言. 形式语言理论主要以无限的语言为研究对象. 例如，所有由 n 个 a 构成的字（$n \geqslant 1$）组成一个语言 $L_a = \{a, aa, aaa, \cdots\}$，它就是无限的. 因此，研究形式语言遇到的第一个问题就是描述问题，描述的手段必须是严格的，而且必须能以有限的语法描述无限的语言. 利用乔姆斯基倡导的变换文法，语言 L_a 可用如下两条生成规则描述：$\{S \to a, S \to aS\}$. S 是变换文法的出发点，由它可以生成 L_a 中的任何句子. 例如，句子 $aaaaa$ 可以这样推导出来：$S \to aS \to aaS \to aaaS \to aaaaS \to aaaaa$. 推导共分五步. 前四步用了第二条生成规则，第

五步用了第一条生成规则.

严格地说,变换文法定义成四元组$G=(\Sigma,V,S,P)$. Σ是字母表,又称终结符号表. V是变量表,又称非终结符号表. S是出发符号,P是生成规则(又称产生式)的集合. 其中Σ、V和P都是有限集,$\Sigma\cap V=\varnothing$($\varnothing$是空集),$S\in V$. 又令α和β分别表示$(\Sigma\cup V)^*V(\Sigma\cup V)^*$和$(\Sigma\cup V)^*$中的元素,则$p$中所有的产生式都可以写为$\alpha\to\beta$的形式,它的含义是用$\beta$替换$\alpha$. 并且至少在一个产生式中$\alpha=S$. 注意,上面的规定说明了每个产生式的左部至少含$V$中的一个非终结符,而产生式的右部可以是空字符串. 今后简称$(\Sigma\cup V)^*$中的元素为串.

利用文法G生成一个语言的句子时,首先取一个$S\to\beta$形式的产生式,用β代替出发符号S,然后取β中的子串β_1,使β_1等于某个产生式的左部α_2,取该产生式的右部β_2以置换β中的β_1,使β变成β',然后再重复这个过程,直到串中只含终结符为止,此时即得到该语言中的一个句子. 为了得到一个新句子,需要再从S出发,重复刚才的置换过程. 由于置换过程一般不是唯一的,所以可以不断地得到新的句子. 所有这种句子之集合即构成该文法产生的语言. 常用$L(G)$表示由文法G产生的语言.

这样定义的文法称为零型文法. 下面是零型文法的一个例子:

$$
\begin{aligned}
&G_1=(\Sigma_1,V_1,S_1,P_1),\\
&\Sigma_1=\{a,b\},\ V_1=\{A,B,S\},\\
&P_1=\{S\to ASB,\quad S\to b,\quad S\to\varepsilon,\\
&\qquad Ab\to a,\ aB\to b,\ AB\to b\}
\end{aligned}
\tag{6.1}
$$

这个文法的置换过程(今后称为推导过程)有无穷多种(例如第一条产生式可以反复使用任意多次),但产生的语句只有两种:b和ε,其中ε是空串(空串的定义是:对任意字符串α:$\varepsilon\circ\alpha$

$=\alpha\circ\varepsilon=\alpha$,其中$\circ$是串联接符,书写时可省去).

本书中,我们恒用大写字母表示非终结符,小写字母表示终结符,因此,今后只需写出产生式集合便可表示整个文法. 由于零型文法太广,不便于研究,因而乔姆斯基又定义了零型文法类的三个子类,它们一个套一个,分别称为一型、二型和三型文法类.

以$|\alpha|$表示符号串α的长度,对零型文法的所有产生式加限制$|\alpha|\leqslant|\beta|$,即得到一型文法. 下面是一型文法的一个例子:

$$S\to aSAB,\quad S\to abB,\quad BA\to AB,$$
$$bA\to bb,\quad bB\to bc,\quad cB\to cc, \qquad (6.2)$$

这个文法产生的语言是$\{a^n b^n c^n \mid n\geqslant 1\}$,其中$a^n$(或$b^n$,或$c^n$)表示$n$个$a$(或$n$个$b$,或$n$个$c$)的联接,而$a^n b^n c^n$表示$a^n$和$b^n$和$c^n$的联接.

如果要求$|\alpha|=1$,即得到二型文法. 下面是二型文法的一个例子:

$$S\to ab,\qquad S\to aSb. \qquad (6.3)$$

这个文法产生的语言是$\{a^n b^n \mid n\geqslant 1\}$.

如果再要求二型文法中产生式的右部(即β)为aB或a,其中$a\in\Sigma^*$,$B\in V$,则得到三型文法. 下面是三型文法的一个例子:

$$S\to 0S,\quad S\to 1S,\quad S\to 0,\quad S\to 1 \qquad (6.4)$$

这个文法产生的语言是全体二进制数的集合,可以表为$\{0,1\}^+$,也称为集合$\{0,1\}$的克林闭包,它的定义是:

$$\{0,1\}^+=\{0,1\}^*-\{\varepsilon\}.$$

这四种文法各有自己的别名. 零型文法又称不受限文法,一型文法又称上下文有关文法,二型文法又称上下文无关文法,三型文法又称右线性文法. 其中一型文法的别名无需解释,三型文

法的别名可从产生式的形状直观看出，二型文法叫上下文无关文法是因为它进行推导(置换)时，可将串中的任一非终结符 A 用任一以 A 为左部的产生式的右部置换，而不必考虑 A 在被置换串中的左邻右舍(即上下文)是什么. 例如，若有产生式

$$A \to a, \quad A \to b, \quad A \to c,$$

则串 $aAbAc$ 可被置换成以下诸串中的任意一个：$aabAc$，$abbAc$，$acbAc$，$aAbac$，$aAbbc$，$aAbcc$，$aabac$，$aabbc$，$aabcc$，$abbac$，$abbbc$，$abbcc$，$acbac$，$acbbc$，$acbcc$.

那么，为什么一型文法被称作上下文有关文法呢?这从下面的定理中可以得到答案：

定理 6.1 对每个一型文法 G_1，一定存在另一个一型文法 G_2，G_2 的产生式皆取如下形式：

$$\alpha A\beta \longrightarrow \alpha\gamma\beta. \qquad (6.5)$$

其中 A 是非终结符，α、β、γ 皆为串，使得 $L(G_1)=L(G_2)$.

可以从这样的观点来看 G_2 的产生式：任意一个串中的非终结符 A 当且仅当在如下条件下可被另一个串 γ 置换：A 的左邻是 α 而右舍是 β，这就是上下文有关的含义. 在二型文法中，产生式(6.5)以 $A\to\gamma$ 的形式出现.

因此，可以把式(6.5)看成是一型文法的另一种定义. 今后，如有 $L(G_1)=L(G_2)$的情形(G_1、G_2 是任意文法)，称 G_1 和 G_2 等价.

在详述了四类文法的定义以后，细心的读者会发现，对我们在介绍这四类文法之前说过的一句话："它们一个套一个"，需要作一点小小的修正. 因为一型文法不允许产生式右部为空串 ε，而二型文法却可以. 只有严格地排除了这种例外以后，才可以说二型文法类是一型文法类的子类.

必须注意把文法和它生成的语言严格区分开来. 例如，从文

法上说，若对二型文法加以限制，不允许它含右部为空串的产生式，则得到二型文法类的一个严格子类，但我们却有如下结果：

定理 6.2 若语言 L 能被某个二型文法生成，则 L 一定也能被某个不含右部为空串的产生式的二型文法生成.

换句话说：加了限制以后，该子类生成语言的能力并未被削弱，因而是原类的一个等价类.

等价文法类的另一个例子是：右线性文法类等价于如下的左线性文法类：要求二型文法中产生式的右部为 Ba 或 a. 因之，右线性文法类和左线性文法类统称为正则文法类.

今后，零、一、二、三型文法简单地表示为 URG，CSG、CFG 和 RG. 它们生成的语言分别称为递归可枚举集(这个名称的含义以后再解释)，上下文有关语言、上下文无关语言和正则语言(又称正则集). 简单表示法分别是 RES、CSL、CFL 和 RL. 也可分别简称零、一、二、三型语言.

各类文法及语言之间的关系可以大致上概括如下：

1. n 型文法生成的语言是 n 型语言.

2. 若不考虑右部为空串的产生式，则对 $1\leqslant n\leqslant 3$，n 型文法同时也是 $n-1$ 型文法.

3. 对 $0\leqslant n\leqslant 2$，至少有一个 n 型文法不是 $n+1$ 型文法.

4. 每个 n 型语言都由一个 n 型文法生成.

5. 对 $0\leqslant n\leqslant 2$，至少有一个 n 型语言不能由 $n+1$ 型文法生成.

作为例子，可以证明式(6.2)中的一型文法生成的语言 $\{a^nb^nc^n\mid n\geqslant 1\}$ 不能被任何二型文法生成. 式(6.3)中的二型文法生成的语言 $\{a^nb^n\mid n\geqslant 1\}$ 不能被任何三型文法生成. 但式(6.1)中给出的零型文法尽管是“真正的”零型文法(不是一型文法，更不是二型、三型文法)，它生成的语言 $\{b,\varepsilon\}$ 却极其简单，可直接

由二型文法$\{S\rightarrow b, S\rightarrow \varepsilon\}$生成. 这表明，尽管复杂性低的文法(如三型文法)不能产生复杂性高的语言，复杂性高的文法(如零型文法)却可能产生复杂性很低的语言. 另外，关于不是一型语言的零型语言的例子将在以后给出.

上面我们介绍了乔姆斯基的文法体系，介绍了该体系中的四大类文法. 每个文法可以其生成语言的能力而被看成是一种计算装置. 不过这种计算装置的描述不象图灵机那样是过程性的(即在给定当前状态和当前符号的前提下，告诉你每一步怎么走)，而是说明性的(即只给出一组计算规则，要你自己去推导). 既是计算装置，就可以比较其计算能力. 可以证明：零型文法类在如下两个定理的意义上与图灵机有相同的计算能力.

定理 6.3 给定任意的零型文法G，它生成的语言$L(G)$可被一图灵机T识别.

证明大意：构造一台三带图灵机，第一条带为输入带，上面有$L(G)$中的任一句子g，第二条带上放着G的全体产生式，这条带也是只读的(不写内容)，第三条带的初始内容是G的出发符号S，以后随着推导的进行，S成为一个(可以兼含终结符与非终结符的)串. 由于产生式集是有限的，因此每一步可用的置换(或称推导)方式也只有有限多种. 图灵机的算法可以这样地设计，使它在反复推导的过程中遍历各种选择的可能性. 并且，每当第三条带上的串只含终结符时，即与第一条带上的输入进行比较，如果相同则停机，表示接受该输入句子. 否则抹去这个终结符串，重写S，重新推导. 由于g可以是$L(G)$中的任意句子，这个图灵机能识别的句子全体正好是$L(G)$. 对于非$L(G)$中的句子，它是不会停机的(除非$L(G)$是有限集).

定理 6.4 如果语言L能被一图灵机所识别，那末它也一定能被一个零型文法生成.

看到零型文法类和图灵机有相同的计算能力，我们不禁要问：是否存在和其它各型文法有相同计算能力的计算装置呢？由于各型文法的计算能力是不一样的，它们一个比一个弱，因此，如果存在与一、二、三型文法有对应计算能力的装置，它们一定是对图灵机能力的某种限制，这类计算装置（包括图灵机在内）统称自动机. 事实上，人们已经研究出了各式各样不同于图灵机的计算装置. 这些装置或者和图灵机等价（有相同的计算能力），或者在能力上弱于图灵机. 已经证明：与一、二、三型文法有相同计算能力的自动机确实存在，让我们来一一地给以介绍.

有穷自动机 它有一个有限的状态集和一个有限的（输入）符号集. 从初始状态开始，它每一次都根据当前状态和当前（输入）符号来决定进入哪个新状态，直到进入终结状态为止. 它的控制函数可以表为

$$(\text{状态},\text{符号}) \longrightarrow (\text{状态}). \tag{6.6}$$

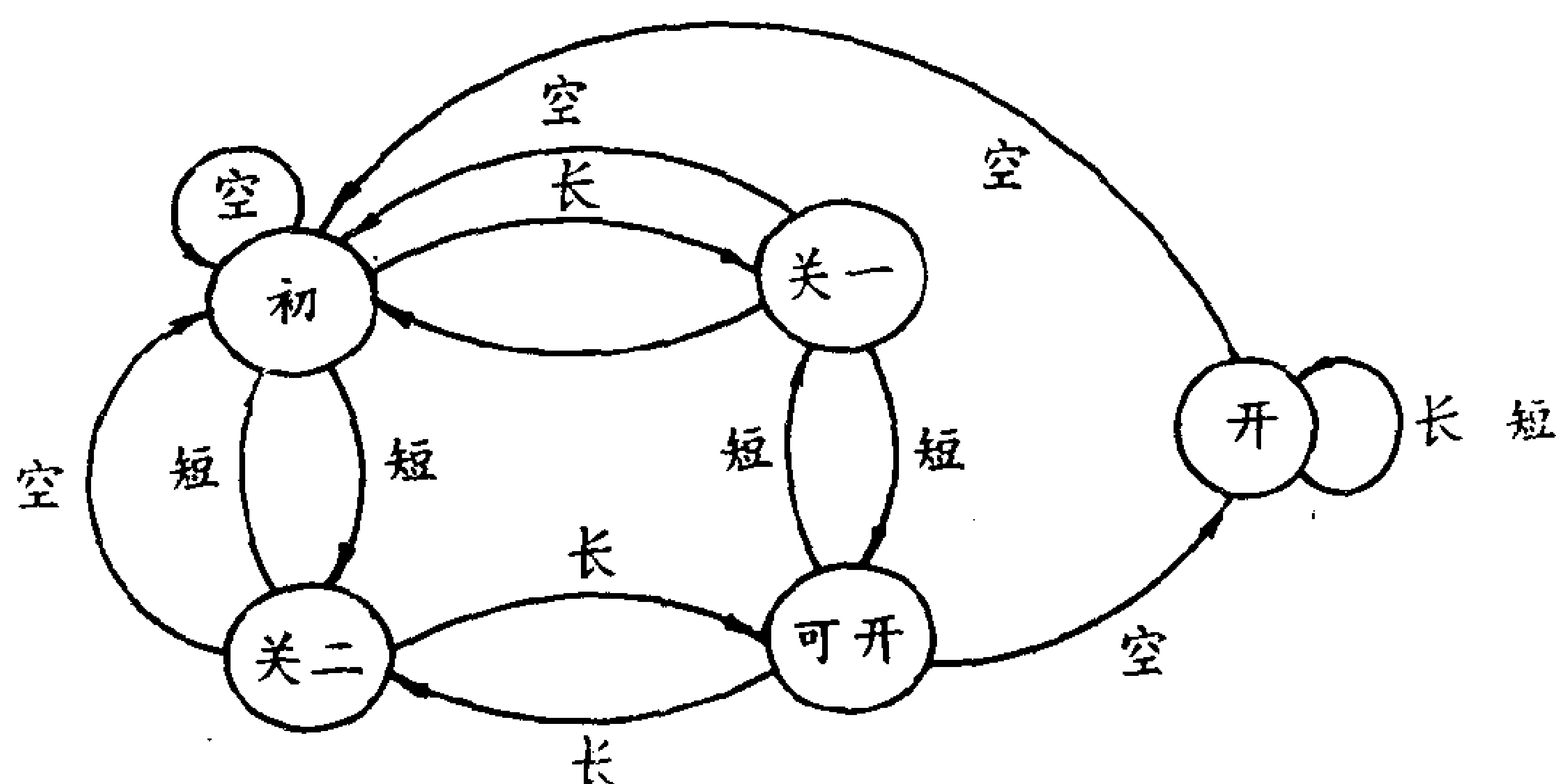
有的有穷自动机也可以输出符号，它带有一个有穷的输出符号集. 相应的控制函数是：

$$(\text{状态},\text{入符}) \longrightarrow (\text{状态},\text{出符}). \tag{6.7}$$

因此，有穷自动机与图灵机的主要区别是它没有移动操作，并且输出的符号不会再行输入. 后面一个区别的含义是：输出的符号不会再次利用. 也就是说，有穷自动机没有存储的概念.

举一个实例，设计一个电铃开门器. 来访客人可给三种信号：长声、短声和无声. 主人规定密码：只有在连续给 $2n$ 个有声信号然后接无声信号时门才自动开启，其中 n 是奇数，并且 $2n$ 个有声信号中包括 n 个长声和 n 个短声（任意排列），则实现此电铃开门器的有穷自动机如第 58 面的图所示，其中圆圈内是状态，弧的旁边是输入符号.

现在考察有穷自动机加上存储功能以后的计算能力. 像图



灵机一样，这里也是利用一条带子作存储符号之用. 但是不应允许在带子上随便存和随便放，否则，就要和图灵机等价了，也就是得不出新类型的自动机了.

通常规定的符号存取方式称为栈，其特点是：先存的符号后用，后存的符号先用. 古人云："譬如积薪，后来居上". 先砍的柴堆在下面，后砍的柴堆在上面，使用时，先烧上面的，后烧下面的. 又如一个人吃菜喜欢吃新鲜的，尽管天天买菜塞在冰箱里，吃的时候总是先吃最后买的. 这种栈常称为下推栈，具有下推栈的自动机称为下推自动机.

举一个实例. 利用下列下推自动机可以识别语言$\{a^nb^n|n\geqslant 1\}$，它的控制函数如下面所示：

1. 初始状态 q_0：输入符号为 a 则进栈，并进入状态 q_1，否则拒绝输入.

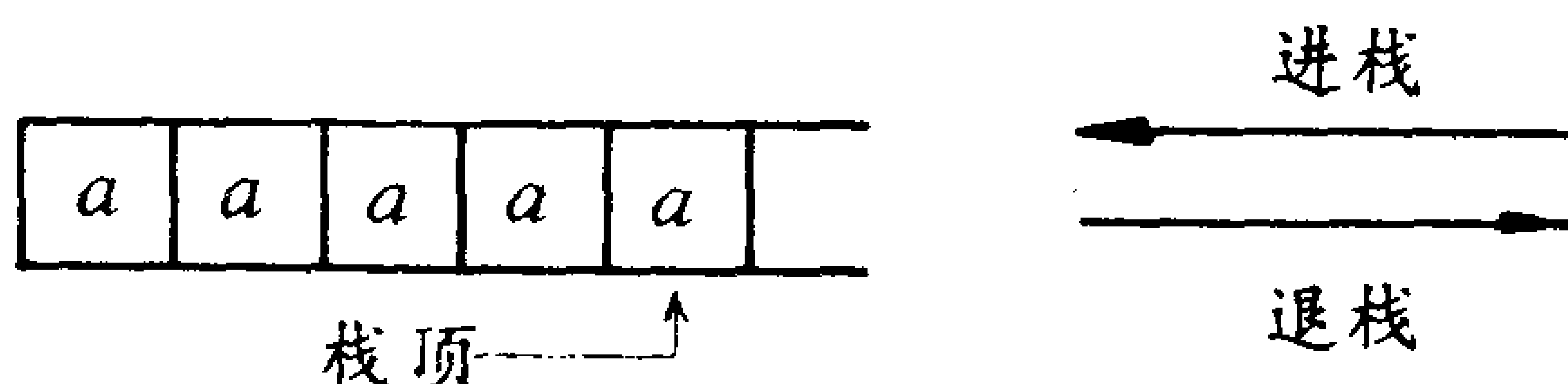
2. q_1：输入符号为 a 则进栈并进入状态 q_1，为 b 则从栈顶退出一 a 并进入状态 q_2，否则拒绝输入.

3. q_2：输入符号用尽且栈顶为空则进入状态 q_3，输入符号为 b 且栈顶不空则从栈顶退出一 a 并进入状态 q_2. 其它情况皆拒

绝输入.

4. q_3:识别成功,输入符号串被接受.

栈的示意图如下:

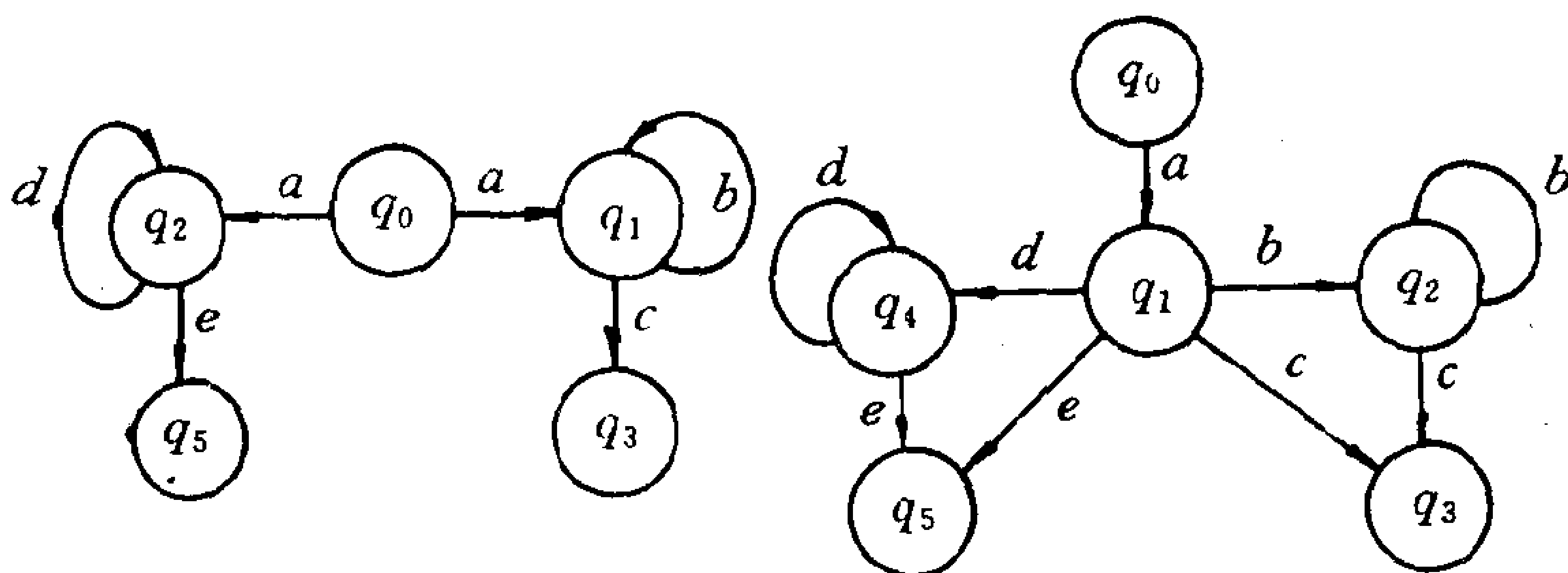


下面要引入我们感兴趣的第三类自动机,称为线性有界自动机.对图灵机读写头的移动操作给以限制,不许它移出带上的输入数据区之外,就得到线性有界自动机.该类自动机获得这个名字的原因是由于下列定理:

定理 6.5 如果把线性有界自动机的上述定义放宽为"不许它移出带上的输入数据区经某个事先给定的线性函数放大以后的区域之外",则它的计算能力不会增强.

例如,设 k 为任意给定的正数,则把读写头移动范围限制在 l 个格子内或 $k\cdot l$ 个格子内,其计算能力是一样的.这里 l 是输入数据区的长度.

在把上述三类自动机和乔姆斯基的文法类给出对应关系之前,还需要补充引进一个十分重要的概念,即非确定型自动机的概念.给定当前状态和当前输入符号以后,如果自动机的控制函数是多值的,亦即它有不止一种操作可供选择,有不止一种新状态可供进入,则称此自动机为非确定型的.对于这种自动机来说,只要存在一条操作序列,使输入句子被接受,即认为此输入句子是被接受的.例如,下面两个有穷自动机接受同样的语言 $\{ab^nc, ad^me \mid n\geqslant 0, m\geqslant 0\}$,但左面那个是不确定型的,而右面那个是确定型的.其中 q_0 是初始状态,q_3 和 q_5 是终结状态.



现在可以给出本章中最重要的定理：

定理 6.6 零型、一型、二型和三型文法的计算能力分别相当于非确定型的图灵机、非确定型的线性有界自动机、非确定型的(有一个下推栈的)下推自动机和非确定型的有穷自动机.

前面给出的两个定理是本定理的特殊情形. 注意这里用到的都是非确定型的自动机，那么它们和确定型自动机的关系如何呢? 我们有：

定理 6.7 确定型和非确定型图灵机的计算能力相当.

定理 6.8 确定型下推自动机的计算能力小于非确定型下推自动机的计算能力.

定理 6.9 确定型和非确定型有穷自动机的计算能力相当.

这里唯独没有提到确定型线性有界自动机及其非确定型之间的关系. 从定义可知，前者的计算能力肯定不会超过后者. 但到底是等于还是小于后者? 这可是困惑了数学家们已数十年之久的一个难题，至今尚未解决.

下面的定理可以作为上述知识的补充.

定理 6.10 带两个下推栈的有穷自动机在计算能力上等价于图灵机.

第七章　计算机和高级语言

——计算能力相同，万变不离其宗

从第二章到第六章，我们给出了五种理论计算模型，研究理论是为了解决实际问题，理论模型最后要落实到实际的计算模型上来，这就是各式各样计算机以及在计算机上运行的计算机语言. 这一章，我们要说一说实际的计算模型.

实际的计算模型首先是电子计算机的体系结构. 现代电子计算机的基本设计思想来自著名数学家冯·诺依曼. 这与 1945 年电子计算机刚诞生的时候有很大的不同. 第一台电子计算机艾尼亚克的运算需要由人来控制. 为了算一道题，要由人事先把完成加、减、乘、除等各类操作的计算部件像搭积木一样地搭起来. 如果换一道题，又得把这些部件分解开来，根据新的要求重新组装，构成新的解题布局，这是非常浪费时间的. 有人注意到，在那种计算机上算一个幂级数的前七项，准备工作至少需要 15 分钟，而计算只需 1 秒钟.

冯·诺依曼和他的同事们发现了这个阻碍计算机充分发挥其作用的问题，他们在 1946 年发表了一份报告，其中提出不仅应该把待计算的数据放在计算机里（这相当于图灵机带子上的

数据)，而且应该把程序(计算机解题步骤，相当于图灵机的控制函数)也放在计算机里，这样，换算一道新题时就不需要采取改变计算机结构的“硬”办法，而只需采取改变计算机中的程序(这是广义意义下的数据)的“软”办法. 冯·诺依曼的思想为英国数学家威尔克斯所实现. 他领导研制的计算机埃特沙克克服了艾尼亚克“执行快，计划慢”的矛盾，在技术上(因此也在计算速度上)来了个飞跃. 由于这个原因，今天的计算机常被称为存贮程序计算机，或干脆称为冯·诺依曼计算机.

冯·诺依曼计算机的基本原理并没有背离图灵机，只是根据实用要求作了重新设计. 它由五个基本部份组成，即运算器、控制器、存储器、输入器和输出器. 事实上，存储器至少还分为内存储器和外存储器两部份. 这样，计算机的最基本部件就分成了运算、控制、内存、外存、输入和输出六部份. 前三部份通常称为主机，其中前两部份合称中央处理器，最后三部份通常称为计算机的外部设备. 需要说明的一点是：由于计算机技术发展极其快，我们不可能详细描述今日计算机的各种技术细节. 读者只能把我们上面说的看成是计算机体系结构的一种最简单模型.

需要计算机算题时，把所需的数据(例如 a、b、c)和算题的程序(例如，根据公式 $(-b\pm\sqrt{b^2-4ac})/2a$ 编成的程序)按规定格式准备好. 其中程序写成一连串指令的形式，指令是计算机能做的最基本的操作. 指令之于计算机，犹如琴键之于钢琴，或图灵机的基本操作(读写，移动，改变状态)之于图灵机. 计算机被启动后，首先由输入器把数据和程序输入计算机内，放入内存储器或外存储器中. 开始算题时，控制器把所需的数据和程序调入内存储器中(如果是在外存储器中的话，否则不需这一步)，并逐条取出指令交给运算器去执行，运算的结果则由输出器输出给用户.

人们常把计算机称为电脑，其实，由运算器、控制器和存储器构成的主机才是真正的电脑，其中存储器起着记忆的作用，运算器和控制器起着思考和解决问题的作用．存储器划分为许多单元，每个单元都有地址．每个单元能放多少个二进位称为这台机器的字长，字长通常为 2 的整数次方．例如，用于控制洗衣机的微电脑字长一般为 4，早期的微型机字长一般为 8，小型机字长一般为 16，现在微型机字长起码是 16，高档微机的字长达 32．每 8 位称一个字节，每一个或两个字节放一个数或一条指令．计算机的存储容量以 K 字节作为单位计算，K＝1024．PC 机（即个人计算机，它是目前国内外使用最广的微机）的内存容量均在 512K 或 640K 字节以上．

根据诺依曼的建议，计算机中的数均表示为二进制的形式．一个二进制数的每一位是 0 或 1．二进制数和十进制数的主要区别是：把逢十进一改为逢二进一，例如，十进制数 228 化成二进制便是 11100100．实际上，不但是通常意义上的数，就连程序、符号等一切需由计算机处理的对象（统称数据）也都以二进制数的形式存放在计算机里．用二进制表示的程序称为机器语言的程序，它的全体指令构成的集合，就是这台机器的机器语言，不难想象，用机器语言来编写程序是一件极其繁琐且极容易出错的事．为了克服这个困难，人们又设计了各种比较高级或相当高级的语言来代替机器语言．这种语言比较接近人在日常生活中或数学计算中使用的语言，摆脱了机器语言的各种细节，因而使用方便，深受用户的欢迎．但计算机只懂得它自己的机器语言，为了用高级语言同计算机打交道，必须找一个“翻译”，把用高级语言写的程序翻译成计算机能懂的机器指令程序，这种翻译本身也是一个程序，称为编译程序．

为了说明高级语言是一个什么样子，本章向读者介绍一种

非常简单而又相当流行，拥有大量用户的高级语言，即 BASIC 语言. 一个 BASIC 程序由一串 BASIC 语句组成. 粗略地说，每一个语句就是一步. 计算机按照先后次序逐一执行各 BASIC 语句. 当执行到语句 END 时，它知道任务已完成，于是结束运行，等待算下一个题目.

一个 BASIC 语句的构造分成三部份：句帽、句头、句身. 句帽是一个非负整数，指示该语句的所在位置，称为标号. 除非在程序中特别指明改变执行次序，否则总是按标号从小到大地执行各语句. 句头是一个英文单词，表示本句要做什么事情，它称为该语句的关键词. 句身就是该语句要做的事. 举例来说：

428　LET　$X=1992\times2+28$

就是一个完整的语句. 标号为 428，关键词是 LET，一般表示要求执行某种计算，而句身 $1992\times2+28$ 是计算的具体内容. 计算结果存入 X 中，X 称为变量.

如果我们要继续计算 x 的平方，则可以再写一个语句

432　LET　$Y=X\uparrow2$,

其中 ↑ 是乘幂符号，表示把 X 的平方值算出来后存入 Y 中，Y 也是变量. 执行完这个语句后 Y 的值已算出，但仍在计算机里，肉眼是看不见的. 为了要知道它是什么，必须要求计算机输出这个值. 语句

435　PRINT　Y

可以完成这个任务.

容易看到，仅仅有 LET 语句是很不方便的. 比如要计算从 1 到 10000 之间每个数的平方值，如用上述办法，至少要写一万个句子！这怎么行？较好的办法是用程序告诉计算机："请算一万次平方值，第一次算 1 的平方，以后每次加 1，一直算到 10000 的平方为止."这句话用程序写出来就是：

```
439  FOR  I=1  TO  10000
440  LET  Y=I↑2
441  PRINT  Y
442  NEXT  I
```

第一句是 BASIC 中的循环开始语句，第四句是循环结束语句. I 叫做循环变量，它指示循环从哪里开始($I=1$)，到何时为止($I=10000$). 四个句子合在一起构成一个循环.

也可以换一种方式写:.

```
455  LET  I=1
456  LET  Y=I↑2
457  PRINT  Y
458  LET  I=I+1
459  IF  I<10001  THEN  455
```

这里又出现一种新的语句，它叫条件语句，其关键词是 IF. 标号为 459 的语句的意思是：当计算机执行到这里时，如果变量 I 的值小于 10001，则离开正常的执行路线(我们在前面说过，正常的执行路线应该是按次序逐句执行的)，转去执行标号为 455 的语句. 从语句 458 可以看出，它每被执行一次，I 的值都要增加 1. 执行一万次后 I 的值变成 10001，语句 459 的条件不再满足，于是不再转回到语句 455 处，而是按正常顺序继续执行 459 以后的语句.

请注意，切莫把语句 458 中的 $I=I+1$ 看作数学上的相等关系，它只意味着把变量 I 的值加 1，再送回 I 中.

现在提出一个问题：假定某市一至十二月逐月产值为 3.8，4.2，5.3，6.1，5.6，4.8，7.3，4.5，6.4，5.8，4.7，6.5(亿元)，该市计委要统计每季平均值和全年平均值，则可编如下程序：

```
507  LET  A=(3.8+4.2+5.3)/3
```

```
508  LET  B=(6.1+5.6+4.8)/3
509  LET  C=(7.3+4.5+6.4)/3
510  LET  D=(5.8+4.7+6.5)/3
511  LET  E=(A+B+C+D)/4
512  PRINT  A,B,C,D,E
```

这个程序好不好？不算太好. 它只能供该市计委使用，不能供其它地方使用，因为产值不会完全一样. 不仅如此，它还只能供该市当年使用，因为明年的产值也不会完全一样. 比较好的办法是使用输入语句 INPUT.

```
511  INPUT  P1,P2,P3,P4,P5,P6,P7,P8,P9,P10,P11,
            P12
512  LET  A=(P1+P2+P3)/3
513  LET  B=(P4+P5+P6)/3
514  LET  C=(P7+P8+P9)/3
515  LET  D=(P10+P11+P12)/3
516  LET  E=(A+B+C+D)/4
517  PRINT  A,B,C,D,E
```

当计算机执行到 INPUT 语句时，它就会停下来，并显示 INPUT? 字样以提示用户输入. 等用户把 12 个数都输入完毕后再继续计算. 由于避免写出具体数据，这个程序现在已经对所有地方和所有年份都通用了.

有的读者会注意到，512 到 515 这四句形式完全一样，是否能简化一些呢？可以. BASIC 提供一种语句，叫自定义函数. 应用这种功能，本例可写为：

```
518  DEF  FNA(X,Y,Z)=(X+Y+Z)/3
519  INPUT  P1,P2,P3,P4,P5,P6,P7,P8,P9,P10,P11,
            P12
```

```
520  LET  A=FNA(P1,P2,P3)
521  LET  B=FNA(P4,P5,P6)
522  LET  C=FNA(P7,P8,P9)
523  LET  D=FNA(P10,P11,P12)
524  LET  E=(A+B+C+D)/4
525  PRINT  A,B,C,D,E
```

这里 DEF 是关键词，它定义了一个名叫 FNA 的函数，该函数的任务是计算表达式 $(X+Y+Z)/3$. 在语句 520 到 523 中该函数被调用四次，每次用不同的值计算.

再一个问题：假如该市计委不仅要计算总产值的季平均和年平均，还要计算钢铁、水泥、玻璃、化肥、棉纱、电机、自行车等行业的季平均和年平均产值，这个程序是否还能用呢？当然，我们可以使用前面介绍过的循环语句，但这样就必须把所有行业的数据一口气算完. 如果要求的不是一口气算完，而是在计算各行业平均产值的中间穿插以一些其它计算，则循环语句就显得无能为力了. 为了解决这个问题，可以把这段程序改写成子程序，具体说来就是加上一行：

```
526  RETURN
```

从 519 行到 526 行的这段程序称为一个子程序. 子程序只需书写一遍，在整个程序中无论什么地方用到它时只需写一个语句

（标号） GOSUB 519

它叫转子语句，作用是告诉计算机："请执行一遍从 519 语句开始，直到第一个 RETURN 语句（即 526 语句）为止的那段程序."由此可见，子程序是极其有用的一个工具.

这里不打算系统讲授 BASIC，而只是想通过 BASIC 这个例子，介绍以高级语言形式出现的实际计算模型.

最早的高级语言大约诞生于 1945 年，是德国人楚译为他的

Z—4 计算机设计的 Plan Calcul，比第一台电子计算机的出现还要早几个月．在电子计算机上实现的第一个高级语言是美国尤尼伐克公司于 1952 年研制成功的 Short Code．而真正得到推广使用，至今仍在流行的第一个高级语言则是由著名计算机科学家巴科斯设计，并于 1956 年首先在美国 IBM 公司的计算机上实现的 FORTRAN 语言．

早期的高级语言主要适用于科学和工程计算，其代表作有 FORTRAN 和 ALGOL60，均已流行全世界，但 ALGOL60 在美国使用较少，而且自 70 年代以来已逐步走下坡路．计算机应用进入商业和行政管理领域后，出现了 COBOL 和 RPG 等便于商界使用的语言．近年来，这类语言和数据库及电子报表软件等结合，形成了一批更方便使用的所谓第四代语言．自 70 年代开始，简单实用、可靠性强的 PASCAL 语言异军突起，在全世界广泛使用，但进入 80 年代以后，它的地位又逐渐被更实用的 C 语言所取代．大型语言的特点是功能齐全，但早期的 PL/1 和 ALGOL68 都不太成功，70 年代开始设计，80 年代开始使用的 ADA 是新一代大型语言，以军用为主要目标．

据统计，全世界的计算机语言起码有几千种，这许多语言虽然功能各异，但是从可计算性的角度看，它们的计算能力却都没有超过图灵机．实际上，它们的计算能力都等价于图灵机．已经证明，一个计算机语言只要除了赋值语句(即 BASIC 中的 LET 语句)之外，还包括顺序语句(由两个或两个以上的语句顺序组合而成)，条件语句和循环语句，那么它的计算能力即相当于图灵机．这里当然要排除各种技术上的限制，如变量的个数，程序的长度，数据的精度等等．

除了图灵机之外，其它各种理论计算模型也都在实际计算模型中有所反映．以递归函数为例，第三章中讲了定义在整数集

上的递归函数，这并不是一种限制，完全可以相应地研究定义在字符串上的递归函数. 设已给定字母表 $\Sigma=\{a,b\}$，则相应的定义是：

1. 基本的原始递归函数.

(1) 后继函数：

$$\text{cons}a(x)=ax,$$

$$\text{cons}b(x)=bx.$$

(2) 零函数：$\text{nil}(x)\equiv\varepsilon$， ε 为空串.

(3) 单位函数：$v_i(x_1,\cdots,x_n)=x_i$.

2. 施行于原始递归函数之上的操作.

(1) 组合操作（函数复合）.

若已知 $f,g_1,\cdots,g_n$ 是原始递归函数，则

$$h(x_1,\cdots,x_m)=f(g_1(x_1,\cdots,x_m),\cdots,g_n(x_1,\cdots,x_m))$$

也是原始递归函数.

(2) 原始递归操作.

若已知 f、g_1、g_2 是原始递归函数，则通过下列方式定义的 h 也是原始递归函数.

a) 当 $n=1$ 时，

$$h(\varepsilon)=w, w\in\Sigma^*, \quad \text{任意}$$

$$h(ax)=g_1(x,h(x)),$$

$$h(bx)=g_2(x,h(x)).$$

b) 当 $n>1$ 时，

$$h(\varepsilon,x_2,\cdots,x_n)=f(x_2,\cdots,x_n),$$

$$h(ax_1,x_2,\cdots,x_n)=g_1(x_1,h(x_1,\cdots,x_n),x_2,\cdots,x_n),$$

$$h(bx_1,x_2,\cdots,x_n)=g_2(x_1,h(x_1,\cdots,x_n),x_2,\cdots,x_n).$$

3. 部分递归函数.

(1) 原始递归函数都是部分递归函数.

(2) 组合操作和原始递归操作施行于部分递归函数后得到的仍是部分递归函数.

(3) $f(\varepsilon)=\perp$(表示无定义)是部分递归函数.

(4) 令 $p(y,x_1,\cdots,x_n)$是一个只取真、假和$\perp$(无定义)三种值的部分递归函数(又称部分递归谓词),函数 f 定义为:

$$f(y,x_1\cdots,x_n)=\text{若 } p(y,x_1,\cdots,x_n)\text{取真值则为 } y \text{ 否则为 } f(\text{next}(y),x_1,\cdots,x_n),$$

则 $h(x_1,\cdots,x_n)=f(\varepsilon,x_1,\cdots,x_n)$是部分递归函数,其中 next$(x)$ 表示在 Σ^* 中的按字典排序仅次于 x 的那个字.

不难看出,上述定义与定义在整数上的极小化操作相当.

著名计算机科学家麦卡锡以字符串上的部分递归函数为理论基础,设计了一种适合于字符处理的 LISP 语言,该语言的计算能力也与图灵机相当,但由于符号处理上的方便,常用于语言文字、乐谱、棋谱及各种符号推理领域,因而有人工智能语言之称.

理论计算模型应用于实际计算模型的另一个例子是数理逻辑.考察具有下列形式的谓词演算合式公式的子集:

$$A_1 \vee A_2 \vee \cdots \vee A_n, \tag{7.1}$$

其中每个 A_i 是一个原子公式,简称原子.原子可以是正的或负的.正原子是普通的谓词,负原子是谓词前加一个非符号,这种形式的公式称为子句.如果限制每个子句中至多有一个正原子,则称为 Horn 子句.

按照谓词演算的等价关系把 Horn 子句改换一个形式,就可以知道它的含义.实际上:

$$\sim P_1 \vee \sim P_2 \vee \cdots \vee \sim P_{n-1} \vee \sim P_n \vee P$$

等价于

$$P_1 \wedge P_2 \wedge \cdots \wedge P_{n-1} \wedge P_n \rightarrow P. \tag{7.2}$$

它表示，如果从 P_1 到 P_n 诸 n 个条件都成立，则 P 也成立. 柯伐尔斯基和柯墨拉瓦尔以 Horn 子句为基础提出了一种新型的计算机语言 PROLOG，这个名字是“逻辑程序设计”的简写. 它的每个语句（称为规则）就是像式(7.2)那样的一个蕴含式，但在书写方式上改为：

$$P:-P_1,P_2,\cdots,P_n. \tag{7.3}$$

当式(7.3)的右部为空时，表示它左部的 P 无条件成立. PROLOG 语言的计算能力也相当于图灵机，限于篇幅，此处不详述 PROLOG 的推导机制，仅举一例以明其大意.

男青年小王发现他的女朋友小张突然不理他了，小王根据平时对小张的了解，赶紧求助于下面的 PROLOG 程序，以搞清小张不理他的原因.

```
dislike(y,x,stingy):—askfor(y,t),don'tgive(x,t,y);
dislike(y,x,habit):—hate(y,t),like(x,t);
askfor(y,t):—like(y,t),don'thave(y,t),talkabout(y,t);
like(小张,TV):—;
like(小王,kalaok):—;
hate(小张,kalaok):—;
```

前三条规则分别表示：若 y 想要 t，x 不把 t 给 y，则 y 不喜欢 x 的吝啬；若 y 讨厌 t，而 x 喜欢 t，则 y 不喜欢 x 的爱好；若 y 喜欢 t 并且没有 t 而又谈论 t，则 y 想要 t.

使用该程序时，小王把？dislike(小张，小王，w)作为目标. w 是变量，其值代表小张不喜欢小王的理由，待定. PROLOG 系统首先把目标和已知事实（程序的最后三行）匹配，未成. 遂把目标和第一个规则的左部匹配，匹配成功，使变量 y，x 和 w 分别取值小张、小王和 stingy，这表示程序首先试探小张是否不喜欢小王的吝啬. 此时原目标化归为该规则右部的两个子目标 askfor

(小张,t)和 don'tgive(小王,t,小张),t 待定. 现在对每个子目标重复刚才的步骤,首先设法使 askfor(小张,t)与已知事实匹配,不成,然后与规则左部匹配,只有第三个规则能匹配成功,使该规则的 y 取值小张. 此时问题进一步化归为 line(小张,t),don'thave(小张,t)和 talkalout(小张,t)三个子目标. 其中 like(小张,t)与已知事实 like(小张,TV)匹配成功,使 t 取值 TV,接着又把 don'thave(小张,TV)与已知事实及规则匹配,均失败,这导致原来的子目标 askfor(小张,t)失败,进一步导致目标 dislike(小张,小王,stingy)失败,说明吝啬不是原因. 于是最初的目标? dislike(小张,小王,w)只好尝试与第二条规则匹配,使 y、x 和 w 分别取值小张、小王和 habit. 接着,两个新的子目标 hate(小张,kalaok)和 like(小王,kalaok)均和已知事实匹配成功,导致原目标 dislike(小张,小王,halit)成功. 于是,小王终于明白了,小张不理他是因为不喜欢他的爱好——kalaok.

第八章　可判定性和可计算性

——国王遗愿为何不能实现？

到现在为止，我们已经介绍了多种不同的计算模型，并探讨了它们的计算能力，看到了六种具有“最高”计算能力的以图灵机为代表的计算装置. 然而即使是像图灵机这样的计算模型也并不是无所不能的. 在本章中，我们要举出一些例子来，说明确实存在着连图灵机也计算不了的问题，这并不是因为图灵机的本事有限，因为这些问题不仅图灵机计算不了，而且任何别的计算模型也计算不了，即它们是不可计算的问题.

让我们再讲一个故事. 古代有个小国，国王酷爱数学. 有一天，西方来了三位使臣，随行的有他们的夫人，国王设宴招待这三对贵宾. 觥筹交错之间，国王忽然心中一动：这三对夫妇的姓名之间似乎有什么联系. 你看，三位使臣的名字是可乐、可口、高乐高果珍. 他们的夫人的名字则分别是乐高、可口可乐可、果珍. 对！如果把第二位使臣的名字，接上第一位使臣的名字（两次），再接上第三位使臣的名字，岂不正好等于第二位使臣夫人的名字，接上第一位使臣夫人的名字（两次），再接上第三位使臣的夫人的名字吗？结果妙极了！

使臣：可口、可乐、可乐、高乐高果珍.

夫人：可口可乐可、乐高、乐高、果珍.

想到这里，国王非常高兴，于是遍询群臣，问谁能找到一个一般的方法，对任何一组夫妇能判断：把丈夫的名字按任意次序连接起来（允许重复），把妻子的名字也按相应次序连接起来，最后是否有可能得到相同的结果？可是这个问题太难了，尽管悬赏黄金千两，良田万亩，全国朝野还是无人能答.

这个问题原来叫波斯特对应问题，该问题的数学叙述是这样的：给定一个有限字母表 Σ，问是否存在一个算法，使得对于任意一组句子对：$(x_1,y_1),(x_2,y_2),\cdots,(x_n,y_n)$，其中 x_i 和 y_i 皆属于 Σ^+，都能在有限步内判定是否存在一种允许重复的拼接方法，使得 m 个 x 句子连接起来正好等于相应的 y 句子的连接，即：

$$x_i x_j \cdots x_k = y_i y_j \cdots y_k. \tag{8.1}$$

如果是先给定了句子对，则答案一般总是可以得到的，前面讲的使臣和夫人的故事是肯定的例子. 也很容易举出否定的例子，但我们需要的是首先给定算法，并要求它对任何的句子对都适用，这就不好办了. 已经证明，这样的算法是不存在的. 无论你用图灵机，或用递归函数，都写不出这样的算法，因此人们称波斯特对应问题为不可判定的. 注意，凡是谈到可判定或不可判定，指的都是一个问题类，而不是单个的问题.

波斯特对应问题是一个著名的不可判定问题. 还有一个不可判定问题，其著名度不在波斯特对应问题之下，那就是图灵机的停机问题. 该问题的叙述是这样的：

定理 8.1 不存在这样的图灵机 U，对于任给的图灵机 T 及 T 的带上的输入数据 I，U 都能在有限步之内判断 T 是否能在有限步之内停机.

显然，要探讨这个问题，首先必须考虑有没有“通用”的图灵机 U，对于任给的图灵机 T 及 T 的带上的数据 I，U 都能模拟 T 的行为. 已经证明，这样的通用图灵机是存在的.

构造通用图灵机的方法大意如下：一台具体的图灵机的操作是由它的控制函数和输入数据决定的. 其中输入数据已以数据形式写在带子上. 如果控制函数也能以数据形式写在带子上，则具体图灵机 T 的行为就完全表示成了数据形式，可以作为通用图灵机 U 的输入了. 事实正是如此. 回想式(2.1)，可知 T 的控制函数可表示为一群如下的五元组：

(原状态，读入符号，写出符号，移动，新状态). (8.2)

不难使五元组的每个成员对应于一个非负整数，于是，任何具体图灵机 T 的控制函数均可表示为一群五元整数组.

通用图灵机 U 的带子上分为三个数据区，其中左面是 U 的工作区域，中间是 T 的控制函数(已表为一群整数五元组的形式)，右面是 T 的输入数据. U 的控制函数按如下的执行步骤设计：

1. 称三个数据区(从左到右)为 D_1,D_2,D_3，其中每个 $D_i(i=1,2,3)$ 有一个特殊符号作为标记，标明当前正在处理的数据. 在算法执行过程中，这三个标记要作相应移动. 下面不再详述.

2. 从 D_2 中取出初始状态，放入 D_1.

3. 从 D_3 中取出当前输入符号，放入 D_1.

4. 把 D_1 中的(状态，符号)对与 D_2 中的诸五元组的头两个元素匹配，匹配不成则停机.

5. 若匹配成功，取该五元组中之写出符号改写 D_3 中被扫描格子的内容.

6. 根据该五元组中的移动操作移动 D_3 中的标记，使它指向新格子.

7. 清除 D_1 内容，并把上述五元组中的新状态写入 D_1.

8. 若该状态是停机状态，则 U 停机，否则转向第 3 步.

由此可以看出，可以模拟执行任何图灵机的通用图灵机是可以构造的. 但是如果被模拟的图灵机不停机，则该通用图灵机亦将不停机. 当然，这个事实不能认为是对图灵机停机问题不可判定性的一个证明，因为这只是通用图灵机的一种构造方法. 反对的人可以说：也许用其它方法可以构造出判定停机问题的通用图灵机呢？

好！现在假定这样的通用图灵机（即可以判定停机问题的图灵机）U 是确实存在的. 假设它有如下功能：任给具体图灵机 T 及其输入 I，U 必定在有限步内作出如下反应：

1. 若 T 能停机，U 也停机，给出结果“1”.
2. 若 T 不能停机，U 停机，给出结果“0”. (8.3)

构造新的图灵机 U'，它与 U 只有一个差别，就是当判断出 T 能停机时，不是也停机并给出结果“1”，而是转入无穷循环（不停机），这一点是极易做到的.

由于 U' 可适用于任意的图灵机，特别地，它也适用于它自己. 另外，图灵机可用任意的数据作为输入，特别地，被判定的图灵机 U' 可以用自身的数据表示 U' 作为输入. 换句话说，我们用通用图灵机 U' 去判断具体图灵机 U' 作用于数据 U' 时的情况. 结果是：

1. 若具体图灵机 U' 能停机，则通用图灵机 U' 不停机.
2. 若具体图灵机 U' 不停机，则通用图灵机 U' 停机并给出结果“0”.

但这两个 U' 实际上是同一台图灵机，因而导致矛盾.

由于判定问题的重要性以及初次接触这个问题的人在概念上常常产生混淆，我们不厌其烦地在下面强调两个注意事项.

第一是不要把数学上某个尚未解决的问题随便看成是判定问题. 例如,费尔马问题是:是否存在 $n>2$,使 $x^n+y^n=z^n$ 有非平凡整数解?哥德巴赫问题是:大于 2 的偶数是否都能表为两个素数之和? 这两个问题的答案要么肯定,要么否定,决不可能在某个时刻被证明为不可判定的,因为它们都不涉及算法. 相反:希尔伯特第十问题:是否存在一个算法可以判定任意丢番图方程有无整数解?却已在 1970 年被苏联数学家马蒂耶塞维奇证明为不可判定的. 一个问题要成为判定问题必须具备以下三个条件:

1. 它是一个问题类,内含无穷多个具体问题.

2. 存在一个特性 p,已知该问题类内满足和不满足 p 的问题各有无穷多个.

3. 要求一个确定的算法,它能对问题类中的每个具体问题,在有限步内回答该问题满足或不满足特性 p.

第二点要澄清的,是一个问题的可判定性不等于它的补问题的可判定性. 我们刚才证明了图灵机的停机问题是不可判定的,指的是要同时做到式(8.3)中的两条是办不到的. 但若只要它做到其中的第一条,则是可以办到的. 办不到的是它的第二条. 这两条互为补问题. 在这个意义上又可以说图灵机的停机问题是半可判定的.

下面一个定理与此问题等价.

定理 8.2 零型文法的识别问题是不可判定的. 即:不存在一个算法 A,使得对任意的零型文法 G 和任意的句子 W(假设字母表 Σ 已给定),A 都能在有限步内判定 W 能否由 G 生成.

与停机问题一样,这个问题的前一半是可判定的. 即确实存在算法 A,使得对任意零型文法 G 和句子 W,只要 W 是由 G 生成的,A 都能在有限步内指出这一点.

图灵机的停机问题和零型文法的识别问题的前一半之所以是可判定的，是因为图灵机的执行步骤和零型文法的推导步骤虽然各有无穷多种变化，但都是可以像自然数一样枚举的. 它们的半可判定性就来自于这种可枚举性（打个比方，如果一把密码锁的密钥是一个正整数，则总可以通过枚举自然数而把密码锁打开）. 由于这一点，零型文法生成的语言（也就是图灵机生成的语言）被称为递归可枚举集.

从数学上看，递归可枚举集是用算法方法（能行性方法）可处理的最一般的对象. 我们甚至有：

定理 8.3 全体图灵机构成一个递归可枚举集.

证明大意：我们在前面已经看到，图灵机的行为由它的控制函数决定，而控制函数可以表为一群五元整数组，把这些五元整数组按某种次序（例如字典次序）排列起来，就得到一个有限的整数序列.

哥德尔提出了一种有效的方法，可以使每个整数序列对应于一个自然数. 该方法的要点是把序列中的第 i 个整数 j 用第 i 个素数的 j 次方代替，并最后把它们乘起来. 例如，序列 2568057 的哥德尔编码是 $2^2 3^5 5^6 7^8 11^0 13^5 17^7$. 由于素因子分解的唯一性，不难对每个自然数判定它是否是某台图灵机的哥德尔编码. 这里留下的二义性是该编码方法不能区分自然数末尾的零. 例如，2566520 与 256652 及 25665200 皆有相同的哥德尔编码，但这是个技术细节，不难通过某种约定解决. 证毕.

设 L 是字母表 Σ 上的语言，则 $\Sigma^* - L$ 称为 L 的补语言. 如果一个语言和它的补语言都是递归可枚举集，则该语言称为一个递归语言，或简称递归集.

定理 8.4 Σ^* 上的一个语言 L 是一个递归集，当且仅当存在一个图灵机 T，T 接受 L 的每个句子，并（以在有限步内停机

并给出拒绝信号的方式)拒绝 L 的补语言的所有句子.

本定理的一个自然推论是:如果一个语言是递归集,那么它的补语言也是递归集.

还有三个问题需要回答:

1. 有没有不是递归可枚举的集合?

2. 有没有是递归可枚举而不是递归的集合?

3. 递归集和乔姆斯基中各型语言的关系如何?

对第一个问题的回答是肯定的.为了说明这一点,我们考察集合:

NTM={x|x 是一个哥德尔编码,并且以 x 为哥德尔编码的图灵机在用 x 作为输入数据时永远不会停机}　　(8.4)

我们已经知道停机问题是不可判定的,因此 NTM 必然不是递归可枚举的.否则,一定有一算法逐一枚举 NTM 中的元素,并拿它与待判定的图灵机作比较,是否停机就会成为一个可判定的问题了.

由此,我们可以构造一个不可计算的函数如下:

$$f(x)=\begin{cases}0, & \text{若 } x \text{ 不是图灵机的编码,} \\ 1, & \text{若 } x\in \text{NTM,} \\ 2, & \text{其它.}\end{cases} \tag{8.5}$$

这个函数是有明确定义的,但却是任何计算模型所不能计算的,因为 x 是否属于 NTM 是一个不可判定的问题.从这里也可以看出,不可计算和不可判定是分不开的两种性质.不可判定性实质上是这样一类不可计算性质,它的计算结果只有两种可能:是或否.

令 TM 为所有图灵机的哥德尔编码的集合,则由前面的讨论可知:TM—NTM 是一个递归可枚举的集合,因为由可以停机的图灵机构成的集合是可以枚举的.我们又知道 NTM 不是

递归可枚举的，于是由前面的定义知，TM—NTM 不是递归集．这回答了第二个问题：确实存在着不是递归集的递归可枚举集．

现在转向第三个问题，我们有：

定理 8.5　每个上下文有关语言都是递归集．

证明思想很简单．由于对 CSG 来说必有 $|\alpha|\leqslant|\beta|$，因此对给定的句子 x 及其长度 $n=|x|$，只需按文法规则逐步推导，直至穷尽所有长度 $\leqslant n$ 的串为止，若 x 在某个串中，则 x 属于此 CSG 生成的语言，否则不是．证毕．

定理 8.6　存在一个不是 CSL 的递归集．

证明大意：给定字母表 $\Sigma=\{0,1\}$，所有以 Σ 为终结符集，且有有限多个非终结符的 CSG 是可以枚举的．接受这些 CSG 产生的 CSL 且对任何输入皆能停机的图灵机也是可枚举的，设这些图灵机为 $M_1, M_2, M_3, \cdots$．规定语言 $L\subseteq\Sigma^*$ 如下：Σ^* 中的成员 w 是 L 中的句子，当且仅当 w 不被第 i 台上述图灵机 M_i 所接受，其中 i 是 w 的二进制编码，于是，L 是递归集，因为任一 w 是否属于 L 可由 M_i 来判定，但 L 不是 CSL，因为 L 不能被任一 M_i 所接受．证毕．

根据上面的两个定理可知，与乔姆斯基层次对应的语言类的包含关系应该是：

$$\mathrm{RL}\subset\mathrm{CFL}\subset\mathrm{CSL}\subset\mathrm{RS}\subset\mathrm{RES} \tag{8.6}$$

其中 RS 表示递归集，其余符号的含义见第六章，这里的包含关系都是真包含．

现在，我们要补充说完第六章中没有结束的故事，在那里曾经谈到两种对立的观点，传统观点认为儿童的语言知识完全是后天学来的，乔姆斯基等人则认为儿童生下来时头脑中就有一部先天的语法，此后通过外界教育而不断加以调整直至确定成一个与环境适应的语法．这个问题可加以数学抽象而变成如下

三个问题：

1. 设 L 是一个语言而生成它的文法不知，问是否能通过枚举 L 中的足够多的句子而把生成 L 的文法归纳出来？（这也是一种计算）

2. 与上面相同，但同时允许枚举不属于 L 的句子（即不仅告诉你哪些句子是对的，还告诉你哪些句子是错的）.

3. 设有两个文法 G_1 和 G_2，如果对 G_1 生成的每个句子 w，都能指出：①w 是否能被 G_2 生成；②若不能，G_2 中与 w 对应的正确句子是什么？能不能把 G_1 逐步改造为与 G_2 具有相同功能的文法 G_3（即 $L(G_2)=L(G_3)$）？

关于第一个问题，有如下结果：

定理 8.7 如果一个语言属于某个包含所有有限语言和一个无限语言的语言类，则该语言的文法在问题一的意义下是不可计算的.

这个结果有利于乔姆斯基.

定理 8.8 在问题二的意义下，上下文无关文法类是可以计算的.

这个结果又不利于乔姆斯基. 然而乔氏学派的人争辩说，儿童一般不大注意成人对他们语言的纠正，因此反倒不起多大作用.

关于第三个问题，有人证明了，在一种比较简单的情况下，文法的调整问题是可计算的. 这是为捍卫乔氏观点而作的一种努力.

在可计算性和可判定性方面已经有了大量的研究成果. 本章的剩余部分将从中选择几项重要的介绍给读者，首先是关于文法和语言方面的研究成果. 前面已经看到，递归可枚举集的识别问题（即零型文法的识别问题）是不可判定的. 除了识别问题

以外，对一个文法 G 还可以提出许多其它问题．如：该文法产生的语言集是否为空集？是否为无限集？是否为全集（即 $L(G)=\Sigma^*$）？若 G_1 和 G_2 是两个文法，则是否有 $L(G_1)=L(G_2)$？或 $L(G_1)\subset L(G_2)$？$L(G_1)\cap L(G_2)$（两个语言之交集）为空？为无限集？等等，等等．这样的问题可以提出一大堆．人们发现，对于递归可枚举集（或等价地：零型文法）来说，它们都是不可判定的．于是有人想到，也许压根儿就不存在对零型文法是可判定的问题．果然，拉伊斯证明了如下的一般性定理：

定理 8.9 如果某个特性 A 是某些递归可枚举集有而另一些递归可枚举集没有的，则 A 对于递归可枚举集来说是不可判定的．

这种特性一般称为非平凡的特性．

对于上下文有关语言，情况只略为好一点点．

定理 8.10 一型文法的识别问题是可判定的，上面提到的其它问题对一型文法都是不可判定的．

定理 8.11 二型文法的识别问题，它产生的语言为空集问题，为无穷集问题，都是可以判定的，除此以外，上面提到的其它性质对二型文法都是不可判定的．

定理 8.12 上面提到的所有问题对三型文法都是可判定的．

最后说几个数理逻辑中的判定问题．

定理 8.13 一阶谓词公式的恒真问题是不可判定的．即不存在一个算法，对任给的一阶谓词合式公式，均能在有限步内判定：该合式公式是否对其中所含变元的任意赋值均取真值．

与图灵机的停机问题一样，此处应把不可判定性理解为半可判定性．即如果合式公式恒真，则总可在有限步内确定之．反之不然．

定理 8.14 只含单变元谓词的一阶谓词公式的恒真问题是可判定的.

单变元谓词指的是 $P(x)$ 这样的谓词，不包括象 $P(x,y)$ 这样的谓词.

推论 8.1 命题演算的恒真问题是可判定的.

第九章　完备性和一致性

——令人想起一个破碎的梦

古代有一个小贩在市场上兜售武器，他说“用我的矛可以刺穿世界上所有的盾，用我的盾可以挡住世界上所有的矛”. 有人问他：“用你的矛刺你自己的盾，结果将如何呢？”小贩无言以对.

这个问题可以表示为如下的一阶谓词演算. 用谓词 penetrate(x,y)表示 x 可以刺穿 y，谓词 resist(x,y)表示 x 可以挡住 y，常量 a 表示小贩的矛，b 表示小贩的盾. 令 A 是世界上所有矛的集合，B 是世界上所有盾的集合. 于是，根据小贩的宣传，我们有

$$\forall\ x\in B,\text{penetrate}(a,x), \tag{9.1}$$

$$\forall\ y\in A,\text{resist}(b,y). \tag{9.2}$$

但根据常识，我们有：

$$\forall\ x,y,\text{penetrate}(x,y)\rightarrow\sim\text{resist}(y,x), \tag{9.3}$$

分别以 b 和 a 例化式(9.1)中的 x 和(9.2)中的 y，得到两个新的事实：

$$\text{penetrate}(a,b),\text{resist}(b,a). \tag{9.4}$$

用式(9.4)的第一个事实例化式(9.3)的左边，就推出

$\sim$resist(b,a)，于是我们同时有 resist(b,a)和$\sim$resist(b,a). 如果一个系统能同时推出 P 和$\sim P$，其中 P 是某个命题，则称此系统为不一致的. 刚才的推导表明，式(9.1)、(9.2)及(9.3)合起来(当然还要加上谓词演算的基本推导规则)就是一个不一致的系统. 小贩卖的矛和盾中，起码有一样是伪劣商品.

现在用 stronger(x,y)表示 x 比 y 强，并增加公式：

$$\forall x,y,\sim \text{resist}(y,x)\rightarrow \text{stronger}(x,y), \tag{9.5}$$

表示若 y 挡不住 x，则 x 比 y 强. 把式(9.3)和(9.5)结合在一起，很容易得到：

$$\forall x,y,\text{penetrate}(x,y)\rightarrow \text{stronger}(x,y), \tag{9.6}$$

上式表示：若 x 能刺穿 y，则 x 比 y 强. 式(9.6)虽然是正确的，但只是我们的直觉，如果要严格地由式(9.3)和(9.5)推出式(9.6)，必须有一个如下形式的推导规则：

$$\frac{P\rightarrow Q,Q\rightarrow R}{P\rightarrow R}. \tag{9.7}$$

它的含义是：若 $P\rightarrow Q$ 和 $Q\rightarrow R$ 皆为真，则 $P\rightarrow R$ 亦为真. 如果在我们的形式系统中缺少了这一条规则，那么，虽然式(9.6)是一个正确的命题，却不能从已知命题(9.3)和(9.5)严格地推出来. 这样的系统称为是不完备的.

一致性和完备性是一个形式系统的两个重要属性. 如果一个形式系统是不一致的或不完备的，那就很难在它上面建立起一套严密的理论体系. 这两个属性和我们前面讨论的问题有密切的关系. 一方面，它们本身构成一个判定问题，即判定一个形式系统是否一致和是否完备的问题. 另一方面，可以把完备性看成是带条件的可计算性，即在给定一组公理和推导规则的前提下能否确定所有取真值的命题.

在形式系统的一致性和完备性方面的最重要的结果，也许

应该是哥德尔的两个著名的不完备性定理. 然而在叙述这两个不完备性定理之前,先说一说发生在 19 世纪和 20 世纪之交的一段数学公案,也许是不无益处的.

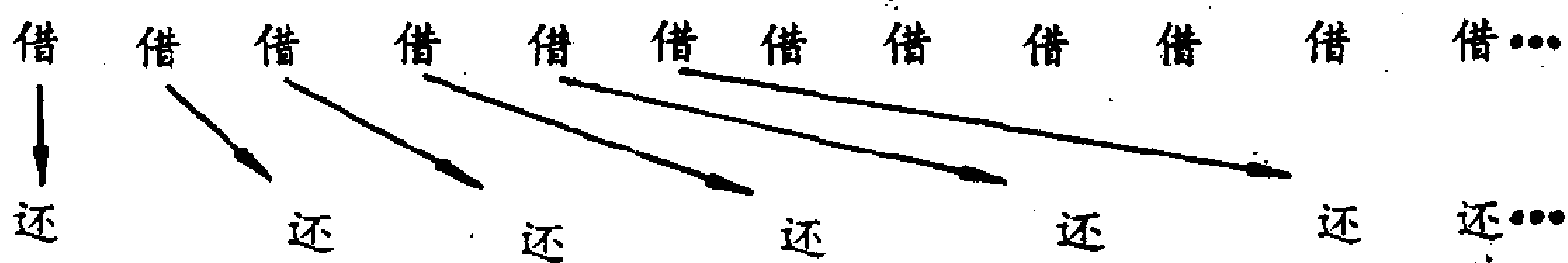
千百年来,许多天才的数学家在为创建一座宏伟、美好的数学大厦而奋斗,这座大厦应该足够地高大,使得从古迄今的一切数学研究成果可以包括在内. 它又应该足够地稳固,使得在大厦内容身的各种数学理论能够互不矛盾,相安无事. 大厦应该有良好的建筑设计,使它能够体现数学内在的和谐与美. 当然,数学大厦不能脱离现实世界,它应该能运用它无穷的智慧来回答现实世界提出的各种问题,为此,数学家们不避矛盾,勇敢探索. 他们战胜了一次又一次数学危机,渡过了一次又一次惊涛骇浪,不断地把数学推向前进. 在这批勇士中最著名的一个,就是著名德国数学家希尔伯特. 他提出了一个建设数学大厦的宏伟纲领,人称希尔伯特纲领,这个纲领最后失败了. 在这一点上,我们可以称希尔伯特是一位失败的英雄. 然而,纲领的失败并不是数学的失败,甚至也不能完全说是希尔伯特的失败,因为他的努力并没有白费. 在为实现纲领所作的奋斗中,希尔伯特和他的同事们留下了一批极为宝贵的遗产,这批遗产至今仍在闪闪发光并指引着后一代的数学家去开拓新的事业.

从直接的意义上说,希尔伯特纲领是为解决一次数学危机而提出的,这次数学危机的根源可以上溯到集合论产生的年代,集合论的创始人康托是一位出生在俄国的德国教授.

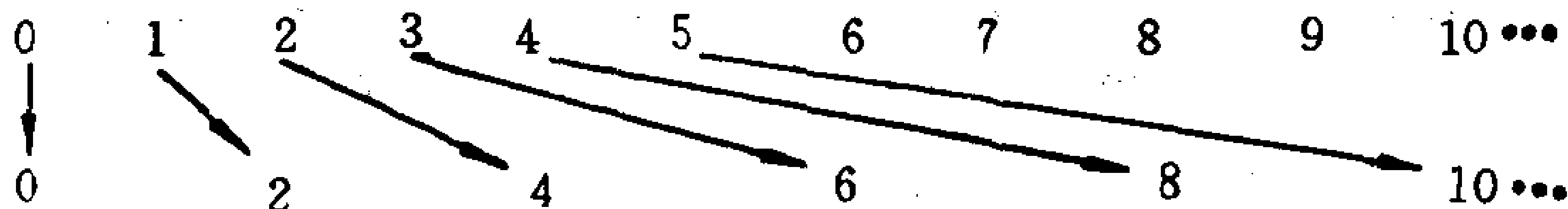
什么叫集合? 集合就是某些对象的全体. 例如,毛泽东的全部藏书,远香楼饭店 231 房间的全部陈设,中国的民间工艺品等都是集合. 这些集合的一部分或把它们并在一起仍然是集合,集合本来是一个极其简单的概念,一个集合中的对象并不需要有什么特殊的性质,它们的唯一共性就是它们都属于这个集合,并

且除了它们之外其它任何对象都不属于这个集合.

假如集合只到包含有限多个对象为止,那也不会有什么矛盾了,问题在于集合论的精华(以及康托的原意)恰恰在于研究包含无穷多个对象的集合,通称无穷集.一进入无穷的领域,就引来了"无穷的"麻烦.例如,要比较两个有穷集合的大小,可以通过比较它们的基数(集合中所含对象的个数)来做到.但是,若要比较两个无穷集合,这个办法就不灵了.康托说,无穷集合也不要紧.如果集合 A 的全体对象能够和集合 B 的某个子集的对象间建立起一一对应关系,而反过来不行,那就称 A 的基数(也叫势)比 B 的基数小.若反过来也行,则称 A 和 B 有相同的基数.这个定义普遍适用于有穷集和无穷集,但在应用于无穷集时,出现了一些一时无法为某些人接受的结论.例如,一个无穷集竟然可以和它的一个真子集有相同的基数.有下列故事为证:孙悟空想开个体饭馆,向南海观音借钱作本.他对观音说,你每年借给我1000元,我保证对每1000元借款,用1500元偿还.南海观音很高兴,把钱借给了他.不久,观音就发现了孙悟空的还钱规律.原来孙悟空只在偶数年还钱.借还关系如下表所示:



从数学上说,这相当于如下的对应关系:



由于孙悟空和观音都是长生不老的,所以孙悟空说的话没

有错，钱确实是每笔都加利偿还了，但观音也没有占到便宜，因为借款和还款之和都是无穷，没法说谁多谁少. 事实上，不仅在借和还之间存在着一一对应关系，而且在借的每一元钱和还的每一元钱之间也存在着一一对应关系. 在无穷的情况下，所谓加利，只是空话.

集合论给数学家提供了锐利的工具. 现在，人们可以说，有理数(两个整数的商)和整数“一样多”，甚至和代数数(整系数多项式的实根)“一样多”，因为它们的基数都一样. 这类集合称为可数无穷集(其元素和自然数 1，2，3，… 一一对应)，但是，所有这些数都比超越数(除去代数数后的实数)“少”，因为前者的基数小于后者的基数，后者称为不可数无穷集. 当然，全体实数也是不可数无穷集.

如果集合论只是给数学带来正面的、积极的成果，那确实很好. 当时许多数学家确实是这样想的. 由于集合论的基本性，人们想把它作为整个数学大厦的最后基石，把整个数学体系的无矛盾性，归结为集合论的无矛盾性. 这样，只要能保证集合论是无矛盾的，那么，建立起一个无矛盾的数学大厦也就有希望了.

可是，偏偏天不遂人愿. 在这块最后的基石上竟然出现了裂纹. 这就是所谓的集合论悖论. 最早发现悖论的人中包括康托自己，后来又有其他人陆续发现了多个悖论. 其中最著名、最易为人理解的是英国数学家罗素在 1901 年发现的悖论. 它的内容是这样的：构造这样的一个集合 A，它的对象的全体就是所有不以自己为对象的集合. 现在要问：A 是否属于 A 自己？如果 A 属于 A 自己，那么 A 就是一个以自己为对象的集合，按规定 A 就不应是 A 的对象. 反之，如果 A 不属于 A 自己，那么 A 就是一个不以自己为对象的集合. 这样说来 A 又应该属于 A. 于是不论哪种情形都会带来矛盾. 这个悖论可以通俗地用一个理发匠的

故事来解释. 假定有这样一位理发匠,他自己约定:只替不给自己理发的人理发. 我们问,他倒底替不替自己理发? 如果他替自己理发,则依上述约定,他不该替这种人(即他自己)理发. 反之,如果他不替自己理发,依上述约束,他又必须替自己理发. 于是陷入了不可自拔的矛盾之中.

悖论的出现,极大地震惊了当时的数学界. 基石上出现裂纹,预示着数学大厦将要倒塌,这可不是什么"杞人忧天"! 当罗素在发现悖论的第二年把这件事通知弗雷格(我们在第五章中已经提到过他)时,后者沮丧地在正要出版的"论数学基础"第二卷末尾添了这样一段话:"一位科学家不会碰到比这更难堪的事了,即在工作完成之时它的基石垮掉了. 当本书等待付印的时候,罗素先生的一封信把我置于这种境地". 另一位数学家狄德金则把正要付印的"连续性及无理数"抽了回来. 一时间,整个数学界似乎都不知道该怎么办才好了.

在人们从最初的惊愕中清醒过来以后,各种各样解决矛盾的主张便提出来了. 以罗素为代表的逻辑主义学派主张把集合论回归到逻辑中去. 例如,毛泽东全体藏书的集合可以用"毛泽东的藏书"这样一个谓词来刻划,使这个谓词为真的个体的集合也就是上面所讲的集合. 如果我们能对谓词实行严格的分级管理,第 $n+1$ 级谓词只允许用第 n 级以下的谓词去定义,那么这也相当于对集合进行了分类,把"集合的集合"之类的言语严格化,就不致于出现像理发匠悖论之类的矛盾.

现在说说解决悖论的第二种主张. 持这种主张的主要代表人物是荷兰数学家布劳威. 他领导的流派叫直觉主义. 顾名思义,直觉主义就是只承认直接感觉到的事物才是真实的. 他们最典型的主张是只承认潜无穷而不承认实无穷. 他们认为许多问题出现的根源在于数学家们往往把实无穷(如康托的无穷集合)

作为客观存在的对象而加以处理．他们认为，只有能在有限步内构造的数学对象才是可靠的．数学上能够承认的无穷只能是正在构造中的无穷（叫潜无穷）．例如，你可以把一个任意大的自然数集合作为对象来处理，却不能把全体自然数的集合作为对象来处理．此外，他们从构造性的观点出发，还坚决反对逻辑上的排中律（非此即彼．即任一命题非真即假，两者必居其一），认为非此不等于即彼．可以用法律为例说明他们的观点．从常识来讲，一个被告要么有罪，要么无罪，法学家在这个问题上分为两派．一派主张"有罪推定"，即若不能证明某人无罪，他就是有罪．另一派主张"无罪推定"，即若不能证明某人有罪，他就是无罪．而直觉主义者却认为不论有罪无罪都应该直接证明．

解决悖论的第三种主张来自形式主义学派，其主要代表人物就是我们所说的勇士希尔伯特．希尔伯特也许是世界上最后一位"全能"数学家，他几乎涉猎了当时每一个重要的数学分支，并且集中精力攻克关键的难题．他在代数不变式、代数数域论、几何基础、狄利赫雷原理与变分法、积分方程与无穷维空间理论，以及物理学、数学基础等领域都有重大建树．根据 19 世纪数学研究的成果及 20 世纪数学发展的趋势，他于 1900 年在巴黎国际数学家大会上提出了著名的 23 个问题．对这些问题的研究极大地推动了 20 世纪数学的发展．希尔伯特不但是卓越的科学家，还是一位正直的学者．他先是拒绝在德国政府为发动第一次世界大战辩护的宣言上签名，后又强烈谴责希特勒的排犹暴行．即使对于自己的老师克隆内克尔，他也持"我爱我师，我尤爱真理"的态度，批判了克隆内克尔排斥一切非构造性数学的观点．

那么什么是希尔伯特的观点呢？希尔伯特最基本的出发点是：古典数学已经取得的成果不能抛弃，因为彻底的逻辑主义（即罗素的主张，称为分支类型论）或彻底的直觉主义（排斥实无

穷，驱逐排中律）都是大部份古典数学的优秀成果所无法满足的. 他坚决反对用大砍大杀古典数学成果的办法来提高数学理论的可靠性，并提出了一套以公理化和有穷推理为核心的主张，具体内容是：

第一：把某个古典数学的基本理论（记为 TH）公理化，再和逻辑演算一起，构成一个形式化的系统 TH_F. 注意它们的区别：TH 是非形式化的，TH 使用的符号表示有具体含义. TH_F 是形式化的，它使用的符号表示没有具体含义.

第二：根据潜无穷的原则，另建一个包含某种初等数论的逻辑系统 TH_m，称为元数学. TH_m 是形式化的，它的符号没有具体含义. 利用 TH_m 作工具，可以用有穷的方法在 TH_F 上进行推理.

第三：他在 TH_F 中允许出现实无穷. 把一切涉及实无穷的命题称为理想命题，而把其它命题称为现实命题. 他认为理想命题虽然不是现实的，却是为了保证数学的完整性和推出有用的数学结果而必需的（如像虚数 i 一样）. 只要在 TH_m 的有穷推理下，TH_F 中不会出现矛盾，这种理想命题即是可以接受的.

希尔伯特纲领的提出有一个过程. 他的数学公理化思想渊源已久. 早在 19 世纪末就作过几何公理化的研究. 他指出应该抛弃几何学中点、线、面的具体含义而只研究它们之间的抽象关系，曾发表过"可以用桌子、椅子、啤酒杯来代替点、线、面"的名言，并于 1899 年完成了几何学基础的研究工作. 进而他又提出了算术公理化的任务，并把它列为 1900 年宣布的 23 个问题中的第二个问题. 1904 年，他在海德堡的一篇讲话中指出单靠逻辑不能构成数学的基础（注意，这否定了罗素的观点），应该建立一个逻辑和算术相结合的系统，这已是证明论思想的萌芽. 1908 年策梅罗对集合论实行了公理化（这是一条分界线. 由康托创立

的包含悖论的集合论称为朴素集合论，由策梅罗开创的集合论称为公理集合论，成为后来数理逻辑四大分支之一），给希尔伯特以极大的鼓舞，推动他进一步发展数学公理化思想，并在1922年的莱比锡会议上初步提出了他的纲领．进一步，他又在1928年的哥尼斯堡会议上提出如下四大任务：分析的无矛盾性；更高级数学的无矛盾性；算术及分析系统的完备性；一阶谓词演算的完备性．这里的无矛盾性即前面所说的一致性．应该指出，在哥德尔冲击波到来之前，他的纲领已取得了部分的成功．

平地一声春雷，惊醒了希尔伯特的数学美梦，出生在捷克的天才奥地利数学家哥德尔，接连向希尔伯特想象中的数学大厦投去几枚重磅炸弹，把它的基石炸得粉碎．并且指出，任何人也休想再建起这样的大厦，因为他已从理论上论证这是不可能的．

其实哥德尔一开始对希尔伯特计划并无歹意，相反，他是想沿着希尔伯特指引的道路去实现这个计划．不料，在实现过程中遇到了不可逾越的障碍．他发现：第一，一个包括初等数论的形式系统 P，如果是一致的，那么就是不完全的．第二，如果这样的系统是一致的，那么其一致性在本系统中不可证．这两条原理被分别称为哥德尔的第一和第二不完备性定理，发表于1930和1931年．

从哥德尔的结论立即可以看出，希尔伯特纲领是不可能实现的．根据第一不完备性定理，希尔伯特的形式系统 TH_F 如果是一致的，那么一定是不完备的，根据第二不完备性定理，用比 TH_F 更基本的元数学系统 TH_M 根本不可能证明 TH_F 的一致性．即使用 TH_F 自己来证自己的一致性也不行，就好像一个人不能拉着自己的头发把自己提起来一样．

勇士希尔伯特是不屈不挠的．冷静思索以后，他发现自己的计划并非已全无希望．因为哥德尔不完备性定理同时也为建立

新型的数学大厦指出了一线光明：需要有比有穷方法更强的方法来论证形式系统的一致性. 事实上，希尔伯特的追随者们正是沿着这个方向，在旧大厦的废墟上重新设计和建造大厦的.

第十章　计算复杂性

——一匹难以驾驭的烈马

在前面，我们把需要计算的问题按不同的角度进行了分类．一种分类方法是根据它们在图灵机意义下是否是可计算的（或是否是可判定的）而分成可计算问题类，半可计算问题类及不可计算问题类．另一种分类方法是根据它们能被具有哪一级能力的计算模型计算而进行分类．由于计算模型五花八门，这种分类体系就不是唯一的了．其中最重要的如图灵机可计算类，线性有界自动机可计算类，下推自动机可计算类，有穷自动机可计算类，等等．在本章中，我们要对问题类进行第三种分类．那就是，在假定它们能被同一种计算模型（主要是图灵机）所计算的前提下，根据计算的复杂性进行分类．有许多种评价复杂性的指标，如用了多少计算时间（按图灵机执行的步数统计），用了多少存贮空间（按图灵机带子上使用的格子数统计），改变了多少次移动方向（图灵机读写头从左移改为右移或相反），移动的总次数（读写头从一个格子移向邻格），回访次数（读写头第一次修改一个格子的内容后返回该格子的次数），预访次数（读写头最后一次修改一个格子的内容前访问该格子的次数），等．其中最重要

的是前两个指标，或称测度，分别称为时间复杂性和空间复杂性测度.

复杂性测度通常用一个函数表示，该函数以图灵机输入数据的长度 n 为变元，一般要求是递归函数，否则不好计算. 分别以 $T(n)$ 和 $S(n)$ 表示时间和空间复杂性. TIME($T(n)$)和SPACE($S(n)$)表示两个问题类，它们分别是在确定型图灵机计算下，时间复杂性不超过 $T(n)$ 和空间复杂性不超过 $S(n)$ 的问题类. 而 NTIME($T(n)$)和 NSPACE($S(n)$)则分别表示在不确定型图灵机计算下相应的问题类. 所谓一个问题类的时间复杂性不超过 $T(n)$，是指其中的任何一个问题，如果输入数据长度为 n 的话，图灵机执行的步数不超过 $T(n)$. 对空间复杂性问题类及其它复杂性问题类的定义与此类似.

在此处要解释一下非确定型图灵机(及其它自动机)的复杂性含义. 在第六章中，我们曾经为这类计算模型的计算能力下过这样的定义：对于它们来说，(从语言识别的观点看)只要存在一条操作序列，使输入句子被接受，即认为此输入句子是被接受的. 那么如何估计它们的复杂性呢？如果存在多条操作序列，使输入句子被接受，即按照最短的那条操作序列来估计它的复杂性. 就好像该图灵机(或自动机)在控制函数取多值的每个分叉路口能猜到当前的最佳路线一样. 特别地，它们不需要像确定型图灵机(自动机)那样用穷举一切可能性的办法来搜索问题的解. 以第六章中的不确定型有限自动机为例，如果输入是 ad^me，它决不会尝试走右边的路去匹配，所以，确定型复杂性类一定是相应的非确定型复杂性类的子类.

被人们研究得最多的，也是最重要的复杂性测度函数有三类：一类是指数型的，常写为 c^n 的形式，其中 c 是常数. 一类是多项式型的，常写为 n^k 的形式，其中 k 为非负整数. 另一类是对数

型的，常写为 $\log n$ 的形式. 具有这些复杂性的问题分别称为指数复杂性，多项式复杂性和对数复杂性. 有时还注明是哪一类测度，如指数时间复杂性，多项式空间复杂性等. 以第二章中的加法图灵机为例，做 $a+b$ 的加法时，输入数据长度为 $n=a+b+3$. 它的时间复杂性为 $T(n)=2n+2$，空间复杂性为 $S(n)=n+2$. 都是线性的. 读者如果做了乘法图灵机的习题，不妨自己计算一下它的时间和空间复杂性.

我们用下列公式来归并几个大的问题类：

$$
\begin{aligned}
&\mathrm{PTIME}=\bigcup_{k\geqslant 1}\mathrm{TIME}(n^k),\\
&\mathrm{NPTIME}=\bigcup_{k\geqslant 1}\mathrm{NTIME}(n^k),\\
&\mathrm{PSPACE}=\bigcup_{k\geqslant 1}\mathrm{SPACE}(n^k),\\
&\mathrm{NPSPACE}=\bigcup_{k\geqslant 1}\mathrm{NSPACE}(n^k),\\
&\mathrm{LSPACE}=\mathrm{SPACE}(\log n),\\
&\mathrm{NLSPACE}=\mathrm{NSPACE}(\log n).
\end{aligned}
\qquad (10.1)
$$

这几大类问题之间已经发现有如下的包含关系：

$$
\begin{aligned}
&\mathrm{LSPACE}\subseteq\mathrm{NLSPACE}\subseteq\mathrm{PTIME}\subseteq\\
&\mathrm{NPTIME}\subseteq\mathrm{PSPACE}=\mathrm{NPSPACE}.
\end{aligned}
\qquad (10.2)
$$

人们感兴趣的是其中的包含关系$\subseteq$能否进一步精确化为$=$或$\subset$(真包含). 研究得最多的是 PTIME$\subseteq$NPTIME 问题，简称 P=NP?问题. 虽有许多计算机科学家和数学家投入了大量的精力，却至今没有解决. 它已被公认为是计算机科学理论中最难最有兴趣的问题之一. 对此，下一章要专题讨论. 目前，多数人倾向于认为 P$\neq$NP.

对于 LSPACE$\subseteq$NLSPACE 问题也作了许多研究. 萨维奇证明了：如果 LSPACE=NLSPACE，则此等号可推广到比对数

大的任意复杂性函数 f(用 $f>\log$ 表示：对任意 $x>0$, $f(x)>\log_2 x$)，对这样的 f 恒有：

$$\mathrm{SPACE}(f(n))=\mathrm{NSPACE}(f(n)). \tag{10.3}$$

可见这个问题的重要.

萨维奇的这个结果的意义还在于，它脱离了单纯研究多项式，指数和对数这三种复杂性的限制，而讨论一般函数 f 的复杂性形式. 关于这个问题，他本人还证明了如下结果：对任意递归函数 f 有：

$$\mathrm{NSPACE}(f(n))\subseteq\mathrm{SPACE}([f(n)]^2). \tag{10.4}$$

用 id 表示恒等函数 $f(n)=n$. 不难看出，此结果与前面的 PSPACE＝NPSPACE(那是关于整个多项式复杂性类的)及 LSPACE⊆NLSPACE(那是关于对数复杂性的，log<id)都没有矛盾. 相反，它是前面结果的细化.

对于式(10.2)中的另一个包含关系 NPTIME⊆PSPACE，彼德逊作了如下的细化，他证明

$$\mathrm{NTIME}(f(n))\subseteq\mathrm{SPACE}([f(n)]^{1/2}). \tag{10.5}$$

根据前一个包含关系，只知道例如非确定型时间复杂性为 n^{100} 的问题，其确定型空间复杂性一定也是多项式型的. 但有了彼德逊的结果，就知道该多项式的次数不会超过 50.

既然允许把任意的递归函数作为复杂性函数，那么自然会产生一个问题，如果 f_1 和 f_2 都是递归函数，$f_1>f_2$，请问以 f_1 作为复杂性上界的问题类一定真包含以 f_2 作为复杂性上界的问题类吗？亦即：是否存在一个问题类，它的复杂性不超过 f_1，但却超过 f_2？如果是这样，那么用递归函数的大小来定义问题类的复杂性分层就是很合适的了.

一般地说，这个结论是不成立的. 事实表明，单单是 $f_1>f_2$ (例如 $f_1\equiv f_2+1$)还不足以使以 f_1 为复杂性上界的问题类超过

以 f_2 为复杂性上界的问题类. f_1 必须"足够地"大于 f_2，才能够和 f_2 拉开一个层次. 下面的结果将表明这一点. 不过在此之前，我们先看一个定性的结果.

定理 10.1 设 f 是一个全递归函数(处处有定义的递归函数)，则一定存在一个递归语言 L，它的识别问题不在 TIME $(f(n))$ 中.

证明大意：按某种原则对所有图灵机进行编号：$M_1, M_2, M_3, \cdots$. 根据第八章中的论证，这是可以做到的. 考虑字母表 $\Sigma=\{0,1\}$，对 Σ^* 内的所有字进行编号：$x_1, x_2, \cdots$，这也是可以做到的. 设计 Σ 上的语言 L 如下：

$x_i \in L \longleftrightarrow M_i$ 在时间 $f(|x_i|)$ 内不接收 x_i. 为了证明 L 是一个递归语言，建立识别语言 L 的图灵机 M' 如下：M' 的功能是，对每个输入 y：

1. 算出 y 的长度 n，并算出 $f(n)$，由于 f 是全递归函数，这一定可以做到.

2. 按编号次序逐个枚举 Σ^* 中的字，找到 $x_i=y$.

3. 根据 i 的值找到第 i 台图灵机 M_i.

4. 模似 M_i 识别 y.

5. 若能在 $f(n)$ 步内识别 y，则拒绝之(y 不属于 L)，否则，接受 y.

显然，图灵机 M' 接受语言 L，因此 L 是递归语言.

现在断言：没有一台图灵机能在时间 $f(n)$ 内识别语言 L. 如若不然，令 $\overline{M}$ 是这样的一台图灵机，它必定在图灵机的编码序列中，设 $\overline{M}=M_j$. 现在问：x_j 在 L 中吗？若 x_j 在 L 中，那么根据刚才的假设，M_j 应能在时间 $f(n)$ 内接受 x_j，但根据 L 的定义，这样的 x_j 不应属于 L；若说是 x_j 不在 L 中吧，M_j 又应拒绝 x_j，根据 L 的定义，x_j 又应在 L 中，所以无论怎么说都是矛盾.

这说明 M_j 不能在时间 $f(n)$之内识别语言 L. 也就是说这样的图灵机根本不存在. 证毕.

从这个定理立即可以推出:存在无穷多个复杂性层次. 我们随便找一个全递归函数 $f(n)$作为起点,根据上述定理必有递归语言 L,它不能在 $f(n)$时间内被任一图灵机识别. 设 $g(n)$是接受 L 的某个图灵机的复杂性,$g(n)$必然是递归函数. 令 $h(n)=\max(f(n),g(n))$,$h(n)$也一定是递归函数,把这个 $h(n)$作为新的一轮的 $f(n)$,再次使用上述定义,又可得到一个复杂性超过 $h(n)$的语言 L',如此循环不已,即得到复杂性的一个无穷层次.

这里说的虽然是时间复杂性,但对于空间复杂性,其结论也是一样的.

严格地说,上面的证明不能看成是构造性的. 因为语言 L 以及新的复杂性函数 $g(n)$和 $h(n)$等都很难用直观的方法表示,我们需要一种更简明的复杂性层次构造方法,为此,引进空间和时间可构造函数的定义.

设 $S(n)$是一个函数. 如果存在图灵机 M,$M\in \mathrm{SPACE}(S(n))$(含义是:对任何长度为 n 的输入,M 占用的带格子数不超过 $S(n)$),并且对每个 n,确有长度为 n 的输入,使 M 占用 $S(n)$个带格子,则称 $S(n)$是一个空间可构造函数.

由于格子数一定是整数,并且每次运行起码占用一个格子,而 $S(n)$之值却不一定是整数,因此,图灵机之空间复杂性为 $S(n)$的实际含义是它的复杂性为 $\max(1,\ulcorner S(n) \lrcorner)$,其中 $\ulcorner x \lrcorner$表示不超过 x 的最大整数.

可以证明,$\log_2 n$,n^k,2^n 及 $n!$等都是空间可构造函数. 并且若 $S_1(n)$和 $S_2(n)$都是空间可构造函数,则 $S_1(n)\cdot S_2(n)$,$2^{S_1(n)}$及 $S_1(n)^{S_2(n)}$也都是空间可构造函数.

定理 10.2 设函数 $S_1\geqslant\log$,$S_2\geqslant\log$,$S_2(n)$是空间可构造

函数，且有

$$\inf_{n\to\infty} \frac{S_1(n)}{S_2(n)}=0. \tag{10.6}$$

其中 inf 表示下极限，则一定存在一个语言 $L\in$ SPACE $(S_2(n))$，但 $L\notin SPACE(S_1(n))$.

类似地，可以定义时间可构造函数，但是时间复杂性和空间复杂性有些不同，它要求构造复杂性层次所用的函数是完全时间可构造的. $T(n)$称为完全时间可构造的，如果存在一个图灵机 M，它对每个长度为 n 的输入，都正好执行 $T(n)$步动作. 相应的定理是：

定理 10.3 如果 $T_2(n)$是完全时间可构造函数，$T_1(n)$是另一个函数，它们满足：

$$\inf_{n\to\infty}\frac{T_1(n)\log T_1(n)}{T_2(n)}=0, \tag{10.7}$$

则一定有语言 L，$L\in$ TIME$(T_2(n))$，但 $L\notin$ TIME$(T_1(n))$.

不难看出，时间复杂性的层次比空间复杂性的层次要稀疏一些. 对后者来说，只要 $S_2(n)$当 $n\to\infty$时的值比 $S_1(n)$的相应值"大得多"(笼统的说法)就可以了. 对前者来说，却要求 $T_2(n)$的值当 $n\to\infty$时比 $T_1(n)\cdot\log T_1(n)$的相应值"大得多"才行. 例如，若 $S_1(n)=n$，$S_2(n)=n\cdot\log n$，则 $S_2(n)$构成比 $S_1(n)$更高的层次. 但若 $T_1(n)=n$，$T_2(n)=n\cdot\log n$，则定理 10.3 的前提不成立，得不出 $T_2(n)$构成更高层次的结论.

对于非确定型的图灵机，也有类似的空间复杂性层次，见下面的定理：

定理 10.4 对 $\varepsilon>0$ 和 $r\geqslant 0$ 有

$$\text{NSPACE}(n^r)\subset\text{NSPACE}(n^{r+\varepsilon}). \tag{10.8}$$

同时，对 $\varepsilon>0$ 和 $r>1$ 有

$$\text{NSPACE}(r^n)\subset\text{NSPACE}((r+\varepsilon)^n). \tag{10.9}$$

注意，如果是确定型图灵机，则相应的包含关系都是前面定理的特例，因为

$$\inf_{n\to\infty}\frac{n^r}{n^{r+\varepsilon}}=\inf_{n\to\infty}\frac{r^n}{(r+\varepsilon)^n}=0. \tag{10.10}$$

可见有关非确定型复杂性层次的结果尚不如确定型复杂性层次的结果那么深入. 另外，有关时间复杂性层次的研究似乎也没有空间复杂性层次的研究那样透彻.

从上面几个定理来看，复杂性层次的区分主要取决于复杂性函数在 $n\to\infty$ 处的行为，复杂性函数的常数因子似乎不起多大作用. 下面的定理为这种直观感觉提供了一定的佐证：

定理 10.5 对于任意正整数 $k\geqslant 1$，任意正数 $c>0$，任意复杂性函数 $T(n)$ 满足

$$\inf_{n\to\infty}\frac{T(n)}{n}=\infty \tag{10.11}$$

者，如果语言 $L\in \mathrm{TIME}_k(T(n))$，此处 TIME_k 表示限于 k 带图灵机，则也有包含关系 $L\in \mathrm{TIME}_k(cT(n))$.

注意我们可使 c 任意小，说明常数因子的作用可以忽略不计. 这个定理的另外一个值得注意之点是它考虑到了图灵机中带子的数目，这是比前面更细致的讨论，因为我们在前面考察的是一般图灵机的集合，没有区分它们的带子数目. 那么，带子数量的增减对图灵机的计算能力究竟有何影响呢？刚才的定理表明：变动常数因子 c 不需要变动带子数目 k. 下面的定理说明增加带子数量与计算能力变化之间的关系：

定理 10.6 若 $L\in \mathrm{TIME}(T(n))$，
则 $L\in \mathrm{TIME}_1([T(n)]^2)$.

定理 10.7 若 $L\in \mathrm{TIME}(T(n))$，
则 $L\in \mathrm{TIME}_2(T(n)\cdot\log n)$.

定理 10.8 若 $L\in \mathrm{NTIME}(T(n))$，

则 $L \in \mathrm{NTIME}_1([T(n)]^2)$.

定理 10.9 若 $L \in \mathrm{NTIME}(T(n))$,

则 $L \in \mathrm{NTIME}_2(T(n) \cdot \log n)$.

布鲁姆发现,各种复杂性测度有一些共同的性质,他把这些共同性质抽象出来,发展了一套公理化复杂性理论,称为布鲁姆测度理论. 他的公理体系如下:

令 $\varphi(n)$是一个部分递归函数. $\Phi(n)$是另一个部分函数,满足下面两个条件:

1. 对所有 n,凡 $\varphi(n)$有定义处,$\Phi(n)$必有定义.

2. 对所有 n、m,$\Phi(n)=m$? 是可判定的.

则 $\Phi(n)$可以看作是 $\varphi(n)$的(抽象)计算复杂性函数. 许多相对于具体的计算复杂性函数的研究成果,可以推广到满足布鲁姆公理系统的抽象计算复杂性函数上来,前面引进的时间复杂性和空间复杂性函数都属于这个范畴. 其中最有兴趣的,是所谓加速定理和间隙定理.

给定一个递归函数 f,是否一定存在一个最佳算法,或曰最佳图灵机,来计算 f? 这里的最佳指的是最低的计算复杂性. 可以是具体的,也可以是抽象的,所谓的加速定理对此给出了一个出乎意料的回答:不一定! 确实存在着这样的递归函数 f,无论你找到怎样好的图灵机 M 来计算 f,总可以构造一个同样能计算 f 的,比 M 的复杂性更低的图灵机 M' !

对于具体的计算复杂性有如下结果:

定理 10.10 存在着这样的递归函数 f,对于任一计算此函数的图灵机 M,如果 M 受控于空间复杂性函数 $S(n)$的话,一定存在着另一个计算 f 的图灵机 M',使 M' 受控于空间复杂性函数 $\frac{1}{2}S(n)$.

附带说明：我们说图灵机 M 受控于某复杂性函数 $\varphi(n)$，如果存在一只依赖于 M 的常数 N，使得当 $n>N$ 时恒有 $g(n)\leqslant\varphi(n)$，此处 $g(n)$ 是 M 的实际计算复杂性函数.

定理 10.11 存在着这样的递归函数 f，对于任一计算此函数的图灵机 M，如果 M 受控于时间复杂性函数 $T(n)$ 的话，一定存在着另一个计算 f 的图灵机 M'，使 M' 受控于时间复杂性函数 $\sqrt{T(n)}$.

布鲁姆用他的公理化复杂性理论对这个问题来了个一锅端. 他证明了如下的一般性结果：

定理 10.12 任给全递归函数 φ，任意指定一种满足布鲁姆公理的测度 B，一定存在一个递归函数 f，使得对于任一计算 f 的图灵机 M，如果 M 受控于 B 中的复杂性函数 g 的话，一定存在另一个计算 f 的图灵机 M'，使得 M' 受控于 B 中的复杂性函数 $\varphi(g)$.

注意这个定理作了两方面的充分推广，一是把具体测度（如空间、时间）推广到任意的布鲁姆测度，另一是把具体的加速函数（如加倍，平方）推广到任意的全递归函数 φ（可以是任意的 n 次方，n 的指数函数，n 的阶乘等等）. 例如：令 $\varphi(n)=2n$，B 为空间复杂性，即得到定理 10.10 的结果. 令 $\varphi(n)=n^2$，B 为时间复杂性，即得到定理 10.11 的结果.

另一个有兴趣的结果是间隙定理. 我们在前面研究计算复杂性层次时就已经发现. 并不是把复杂性函数加大一点点就可以得到新的复杂性层次，而总是要拉开相当的距离才行. 人们有理由问：不能把复杂性层次分得更细吗？是不是仅仅由于我们的证明技巧还不够，才未能做到这一点呢？布洛丁的下列结果告诉我们：复杂性层次不能任意细分，两层复杂性之间的非空间隙是确实存在的. 不仅如此，这种间隙想要多大就多大.

定理 10.13 任给全递归函数 $\varphi \geqslant id$，任意指定一种满足布鲁姆公理的测度 B. 一定存在一个单调的递归函数 f，使得受限于 f 的复杂性问题类即是受限于 $\varphi(f)$ 的复杂性问题类.

选择足够大的 φ，即可得到任意大的间隙.

第十一章　P＝NP?

——一个难倒了无数数学家的谜

我们已经论证过计算机能力的一种极限，就是存在着它“计算不了”的问题类. 在本章中，我们还要论证计算机能力的另一种极限，就是除了“计算不了”的问题类之外，还存在着它“实际上计算不了”的问题类. 这是什么意思呢?从理论上说，如果对计算所需的时间和空间不加限制的话，后一类问题总是可以计算或解决的，因为确实已经知道计算这些问题的具体算法；但从实际上说，这类问题却又是计算机难以计算的，因为这些问题是如此之复杂，需要计算的时间是如此之长，以致于没有等到计算完，计算机已经磨损为一堆废铁甚至尘埃，或许算题的人已经传到他的第八万代重孙，或许甚至地球和太阳系都已经不存在了.

这并不是危言耸听，我们在前面已经谈到了计算有一个复杂性问题. 不过在那里主要是理论上的探讨，不涉及在计算机上作具体计算. 如果考虑到要真刀真枪地用计算机来算，那就必须弄清楚，什么样的复杂性是我们可以容忍的. 换句话说，具有什么样复杂性的问题类被认为是现实可计算的，而超过了这个复杂性界限的问题类就不再是现实可计算的. 为了说明这个问题，

我们再次考虑复杂性的阶，不过这一次不是以图灵机带子上的数据（那是抽象的）作为依据，而是以现实的问题作为依据.

我们用三个例子来说明问题，这三个例子是：求 n 个数的平均；求 n 个数两两相乘的乘积；写出所有的 n 位整数. 这三个例子的复杂性阶各不一样，解第一个例子需要做 $n-1$ 次加法和一次除法，解第二个例子需要做 $C_n^2=n(n-1)/2$ 次乘法，解第三个例子需要写出 10^n 个 n 位数字. 可以看出，第一个例子属于复杂性为线性的问题类. 第二个例子属于复杂性为平方型的问题类，而第三个例子属于复杂性为指数型的问题类，在所有这些例子中，问题本身的规模都以 n 表示.

在实际计算中，不同的复杂性意味着什么呢？有人列出了下面这张表格：

规模 / 复杂性	10	20	30	40	50	60
n	1 秒	2 秒	3 秒	4 秒	5 秒	6 秒
n^2	1 秒	4 秒	9 秒	16 秒	25 秒	36 秒
n^3	1 秒	8 秒	27 秒	64 秒	125 秒	216 秒
n^5	1 秒	32 秒	243 秒	17 分	52 分	130 分
2^n	.001 秒	1 秒	17.9 分	12.7 天	35.7 年	366 世纪
3^n	.059 秒	58 分	6.5 年	3855 世纪	2×10^8 世纪	1.3×10^{13} 世纪

从这张表中，可以明显地看出在多项式型复杂性（表中只列到 n^5）和指数型复杂性（表中只举 2^n 和 3^n 的例子）之间的鸿沟. 当问题规模从 10 上升到 60 时（不过上升为 6 倍！），n^5 型复杂性的计算时间仅从 1 秒上升为 130 分，而指数型的 2^n 类问题的计算时间却从千分之一秒上升为 366 个世纪. 对于 3^n 类问题来

说，计算时间更从 0.059 秒上升为 1.3×10^{13}个世纪. 为了理解这个数字的含义，不妨记住如下的事实：现代人诞生于 20 万年前，相当于 2×10^3 世纪. 古人类诞生于 400 万年前，相当于 4×10^4 世纪. 恐龙灭绝于 6500 万年前，相当于 6.5×10^5 世纪. 地球诞生于 45 亿年前，相当于 4.5×10^7 世纪. 而根据大爆炸宇宙学的推算，我们所在的宇宙的年龄小于 200 亿年，即 2×10^8 世纪. 所有这些漫长的岁月，比起计算一个 3^n 类问题所需的 10^{13}世纪来，却又都是何等短暂的一瞬！

可见，用单纯延长计算时间的办法来求解复杂性高的问题是行不通的. 那么，用提高计算机速度的方法行不行呢？在过去的几十年中，计算机速度在飞速提高，我们在第七章中提到过的第一台电子计算机，每秒钟可做十位十进制数的 5000 次加法，或 500 次乘法，或 50 次除法，相当于普通人计算能力的十万倍. 可现在，人们已经造出了每秒能作数百亿次浮点运算的电子计算机. 平均每十年提高运算速度 100 倍. 既然如此，那就让我们造出更快更大的计算机好了. 遗憾的是，这条路仍走不通. 为了说明这一点，让我们再借用一张别人列出的表格，其中说明了，对具有不同复杂性的问题类，当计算机的速度提高时，解决相应问题类的能力会提高多少.

速度 / 复杂性	现在用的计算机	快 100 倍的计算机	快 1000 倍的计算机
n	N_1	$100N_1$	$1000N_1$
n^2	N_2	$10N_2$	$31.6N_2$
n^3	N_3	$4.64N_3$	$10N_3$
n^5	N_4	$2.5N_4$	$3.98N_4$
2^n	N_5	$N_5+6.64$	$N_5+9.97$
3^n	N_6	$N_6+4.19$	$N_6+6.29$

从表中可以看出，对于复杂性为线性的问题，如果原来在一小时之内可以计算的问题的规模是 N_1，则当计算机速度提高到 m 倍时，解题规模也提高 m 倍，达到 mN_1. 对于复杂性为 n^5 型的问题，计算机速度提高到 100 倍和 1000 倍时，解题规模也分别提高为 2.5 倍和 3.98 倍. 可是指数型的问题就不一样了，对于 2^n 类的问题，即使计算机速度提高 1000 倍，解题能力也不能成倍增长，哪怕增加一倍也不行，而只能是多算约 10 道题. 附带说明一句，这张表和上一张表都是根据实际的计算机运行记录推算的，解算的都是特定的具体问题. 如果换一批问题，数据可能会有小的变化，但不会有本质的不同.

因此，单纯依靠提高计算机速度也不是办法，于是，人们认为，关键问题是要从数学上找出好的算法，而决不能用计算机来硬拚. 果然，计算数学家们找出了各种各样的好办法，使一些本来难以计算的问题变成好计算的了. 例如，用计算机对卫星照片进行处理，如果在一张 10 厘米见方的照片上以一微米（万分之一厘米）为间隙打上格子，则处理一张照片需要进行 10^{16}次运算，即 1 亿亿次运算，即使用每秒百亿次的计算机也要连续算上十多个昼夜. 但是，有人发明了一种好方法，即所谓快速傅里叶分析，把两个 n 阶多项式相乘的复杂性从 n^2 降低到 $n\cdot\log n$. 仅此一项，就使同样的计算机解同样的问题的时间降低到 1/3 秒！

照这样说，我们只需为每个问题类找出巧妙的算法就行了？但是在作这个结论之前应该首先问一下，巧妙的算法永远存在吗？实际上，像傅里叶分析涉及的 n^2 复杂性还远非是我们的大敌，从前面两张表可以看出，真正难对付的是指数型复杂性，它的对立面是多项式复杂性. 如果对每个问题类都能找到多项式复杂性的算法，尽管这种复杂性依然可以很高，我们就已经谢

天谢地了．人们习惯于把理论上可计算的问题类（即图灵机可计算的）称作能行可计算的，而把具有多项式复杂性的问题类称作有效可计算的．后者是前者的一个子类．根据洪加威关于计算模型相似性的研究，我们有理由相信，一个问题类是否是有效可计算的，正像它是否是能行可计算的一样，都不取决于所使用的计算模型（当然是在和图灵机等价的计算模型范围之内）．

怎样才能知道是否对各种问题类都存在多项式复杂性的算法（以后简称多项式算法）呢？从理论上来说，就是要解决 P＝NP？的问题．也就是说，用确定型图灵机在多项式时间内可解的问题类是否即是用非确定型图灵机在多项式时间内可解的问题类？这个问题的重要性在于：迄今为止我们所知道的"有意义"的问题类都属于 NP，如果能够证明 P＝NP，那末这些问题类也就都属于 P，即可以找到多项式算法．当然，其中不应包括我们在本章中提到的那个打印问题，它不能算是一种计算．

在还没有证明 P＝NP 还是 P≠NP 的情况下，我们只好暂时把还未找到多项式算法的问题不归入 P 类．由于 P 肯定是 NP 的子类，因此这些问题可暂时算入 NP—P 中．人们发现，对 NP－P 类中的问题可加以分化，区别对待．为此，首先引进 P 归约的概念．

我们用形式语言的方式来解释 P 归约．假设有两个字母表 Σ 和 Γ，以及定义于其上的两个语言 A 和 B，$A\subseteq\Sigma^*$，$B\subseteq\Gamma^*$．如果存在把 A 映入 B 的一对一映射 f，使得

$$\forall\ x\in\Sigma^*, x\in A\leftrightarrow f(x)\in B, \tag{11.1}$$

并且 f 可用一台确定型的图灵机在多项式时间内计算，则称语言 A 的识别问题可以 P 归约为语言 B 的识别问题．一个直接的推论是：如果语言 B 可以在多项式时间内被识别，则语言 A 也

一定在多项式时间内可识别，有时简称 A 可以归约为 B，用记号 $A\propto B$ 表示。

假设 C 是 NP 中的一个子类，问题类 B 称为是 C 完备的，如果 B 是 C 的一个子类，并且 C 中任何问题类 A 均可 P 归约为 B. 当 C 即是 NP 自己时，称 B 是 NP 完备的. 这种问题类具有特殊的意义，因为任何 NP 类问题均可 P 归约于它. 换句话说，只要能证明其中任何一个是多项式时间可解的，则整个 NP 类也都是多项式时间可解的，P＝NP 的问题也就解决了.

发现第一个 NP 完备问题的是柯克. 他选择的问题是命题公式的可满足性问题，简称 SAT 问题. 问题内容如下：求一个算法，使得对任意一组有限个命题公式（其中每个公式都是用或符号联结起来的命题变量或带非符号的命题变量）都可在有限时间内判定：是否存在对各命题变量的一个真值指派，使得所有这些命题公式均被满足. 这种特殊形式的命题公式称为命题子句，它是第七章中讲过的谓词子句的特例，其可满足问题是一个复杂的组合问题. 例如，子句组 $\{A\vee B, \sim A\vee\sim B\}$ 是可满足的，相应的真值指派是 $A=T, B=F$ 或 $A=F, B=T$. 而子句组 $\{A\vee B, \sim A, \sim B\}$ 则是不可满足的.

柯克证明的大意是对 NP 中的每个问题类加以抽象化，使之成为非确定型图灵机计算的问题，然后构造一个相应的 SAT 问题，并证明前者可以 P 归约为后者. 由于证明太长，不能在此详述其细节. 柯克开了头以后，人们利用他的方法，发现了一个又一个 NP 完备问题，这样的问题现在已至少有好几百个，此处略举数例.

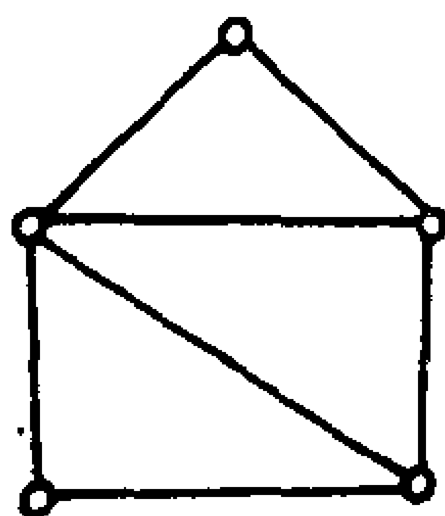
1. 三满足问题. 在 SAT 问题中，把子句的长度（子句内所含原子个数）限制为 3，称为 3SAT 问题，它仍是 NP 完备的.

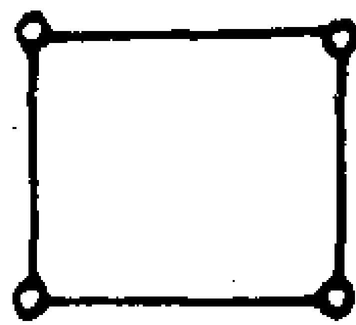
2. 子图同构问题. 对任意两个图 A 和 B，问 B 是否同构于

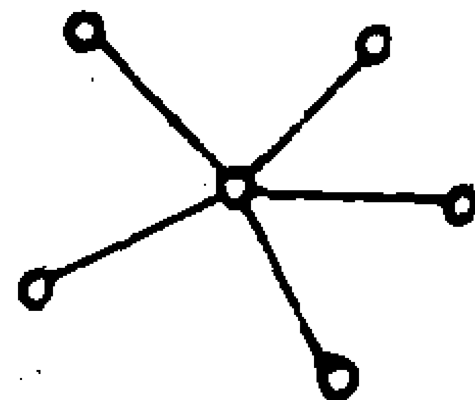
A 的一个子图？

此处的图是一个数学概念，具体指无向图．它由一组节点 V 和一组边 E 组成，其中每条边连接 V 中两个不同的点，并且每两个点之间最多有一条边相连．图 G 的一个子图 G' 由 G 的节点集 V 的子集 V' 及 G 的边集 E 的子集 E' 组成，E'中的边只联结 V' 中的点并且不改变原来的联结关系．两个图 G_1 和 G_2 称为同构，如果在节点集 V_1 和 V_2 及边集 E_1 和 E_2 间分别有一一对应关系，并且满足下列条件：边 e_1 和 e_2 对应，当且仅当 e_1 联结的两个点分别和 e_2 联结的两个点对应．

在下图中，图 B 和图 A 的一个子图同构，图 C 和图 A 的任何子图都不同构．


A


B


C

3. 三维匹配问题．q 为正整数，在三维空间中给定立方体 $0 \leqslant x, y, z \leqslant q-1$．任给它的一个子集，问子集中是否存在 q 个整数格点，其中任意两个格点的所有座标值都不一样？

4. 哈密顿圈问题．任给一个图，能否在图中走一圈后回到出发点，并且除出发点外，其它每个点恰好经过一次？

在几百个 NP 完备问题中，绝大多数都以上面这种判定问题的形式出现，但也有以最优化问题的形式出现的，其中比较著名的一个是旅行推销员问题（也称货郎担问题）．它的大意是：任给 n 个城市及每两个城市之间的距离，求一条路程最短的环

路，它经过每个城市恰好一次，并回到出发点．注意这不同于哈密顿圈，在这里，每两个城市间都有道路直接相连．

旅行推销员问题可以换成如下的判定问题：任给一整数 $k>0$，问是否存在路程长度不超过 k 的环路？这个问题的复杂性不会超过原来问题的复杂性．因为如果最短环路找到了，那么长度不超过 k 的环路的存在性问题也就解决了．已经证明这个判定问题是 NP 完备的．

有一些问题类，它们和另一些问题类之间似乎只差毫厘，但却有本质区别，一方属于 P 类，而另一方属于 NP 类，例如：

1. 假定图的每条边上都标明长度，找两点之间的最短通路是 P 类问题，而找两点之间的最长通路是 NP 完备类问题．

2. 给定正整数 k，问图中是否有不超过 k 条边的集合，使每个点至少接在一条边上，这是 P 类问题．但是问图中是否有不超过 k 个点的集合，使每条边至少接在一个点上，这是 NP 完备类问题．

3. 前述三维匹配问题属于 NP 完备类，但二维匹配问题（在正方形，$0\leqslant x$，$y\leqslant q-1$ 的子集内找符合同样条件的 q 个整数格点）却属于 P 类．

4. 前述哈密顿圈问题属于 NP 完备类，但欧拉圈问题（在图中走一圈回到出发点，要求每条边恰好经过一次）却属于 P 类．这有赖于欧拉证明的下列事实：一个图具有欧拉圈的充分必要条件是该图连通并且每个点与偶数条边联接．

还有一些问题类，既未能证明它们属于 P 类，也未能证明它们是 NP 完备的．例如，任给两个图，判断它们是否同构的问题．又如判定一个数是否合数（大于 2 的非素数）的问题．米勒在 1976 年给出了一个判定正整数是素数还是合数的算法，并且证明了，如果数论中的广义黎曼猜想（此猜想涉及知识太多，

在此不易说清）成立，那末该算法属于P类.

既然有一些问题未能证明为属于P类，也未能证明为属于NPC类（即NP完备类），那么是否有可能事实上它们确实既不属于P类，又不属于NPC类？亦即 $NP-(P\cup NPC)\neq\varnothing$？我们称这些问题所属的类为NPI类（意为：非NP完备类）. 下列定理有助于解答这个问题.

定理 11.1 令 B 是一个不属于P类的递归语言，则存在一个P类语言 D，使得语言 $A=D\cap B$ 不属于 P. $A\propto B$，但 $B\propto A$ 不成立.

从这个定理可以推出，如果 $P\neq NP$，则集合NPI一定非空. 因为：设 N 是NPC中的一个递归语言，如果 $P\neq NP$，则 $N\notin P$，于是 N 满足定理中语言 B 的条件. 同时，由于 $N\propto A$ 不成立，必有 $A\notin NPC$. 又由于 $A\notin P$，所以只能是 $A\in NPI$，即NPI非空.

不仅如此，上述定理实际上还表明：在NPI内部还可以分出无穷多个层次来，为此只需把定理中构造出来的 A 作为新的一轮的 B，马上可以看出一定存在 A'，使 $A'\propto A$ 而 $A\propto A'$ 不成立，并且 $A'\notin P$. 这个构造过程可以无休止地继续下去.

进一步，雷特纳证明了，如果 $P\neq NP$，则存在语言对 $C, D\in NPI$，使得 $C\propto D$ 和 $D\propto C$ 均不成立. 这表明NPI所含的问题类构成一个偏序集（偏序的定义见第十二章）.

另一个有趣的问题是 P 同构问题. 我们已经知道NPC类中的问题是可以互相 P 归约的. 但归约用的多项式时间函数不一定一样. 如果 A 可以通过函数 f 而 P 归约到 B，B 又能通过 f^{-1} 而 P 归约到 A. 并且 A，B 两个问题类的问题在归约 f 下一一对应，则 A 和 B 称为是 P 同构的. 哈特曼尼斯等人发现：现在已经知道的所有NPC类问题之间都是 P 同构的. 可惜我们

还没有找到全部 NPC 类问题，否则，证实全部 NPC 类问题之间有 P 同构关系将导致结论 P≠NP. 因为如果 P=NP 的话，那末所有 P 类问题也都是 NPC 类问题，因之也都互相 P 同构. 但 P 类问题中包含有限语言和无限语言，这两种语言之间是不可能 P 同构的，矛盾！

前已提到，NPC 类问题中包括最优化问题. 为避免陷入指数复杂性，人们有时满足于求一个次优解，这就产生了所谓逼近问题. 以 x 表示某问题的最优解，$A(x)$表示它的近似最优解，m 表示最优性度量函数，则

$$r=\left|\frac{m(x)-m(A(x))}{m(x)}\right| \tag{11.2}$$

称为解 $A(x)$逼近最优解的程度. 给定 $\varepsilon>0$，若对某类最优化问题中的每一个都能使 $r<\varepsilon$，则称此问题类为可 ε 逼近的. 若只用多项式时间算法就可做到 ε 逼近，则称此问题类为 P 可逼近的. 令人迷惑不解的是，虽然 NPC 类中的问题相互间都有 P 同构关系，但只有部份是 P 可逼近，而另一些（如旅行推销员问题）却在 P≠NP 的前提下不是 P 可逼近的. 这说明 P 同构不保持 P 可逼近的性质. 因为人们倾向于相信 P≠NP.

在第八章中我们引进过补问题的概念. 如果 R 表示字母表 Σ 上的语言 L（$L\subseteq\Sigma^*$）的识别问题，则它的补问题 R^c 表示语言 Σ^*-L 的识别问题. 如果 R^c 属于 P 类，或 NP 类，或 NPC 类，或 NPI 类，则我们相应地称 R 本身属于 CO—P 类，或 CO—NP 类，或 CO—NPC 类，或 CO—NPI 类. 随之而来的一个问题是：NP 类等于 CO—NP 类吗？人们猜测，这两个类是不相等的. 因为从 NP≠CO—NP 可以推出 P≠NP. 证明如下：由于 P=CO—P，因此，如果 P=NP 的话，将有如下推导

$$\mathrm{NP}=\mathrm{P}=\mathrm{CO-P}=\mathrm{CO-NP}. \tag{11.3}$$

其中 CO－P＝CO－NP 是 P＝NP 的自然推论.

此外，已经证明了，如果存在 NPC 中的问题 R，使得 $R^c \in$ NP，则定有 NP＝CO－NP. 换句话说，如果问题 R 和 R^c 都属于 NP，则只有在 NP＝CO－NP 的情况下 R 才能属于 NPC. 但根据前一个结论知道 NP＝CO－NP 的可能性很小. 于是我们可以有相当大的把握说，如果 R 和 R^c 都属于 NP，则 R 不是 NP 完备的.

与此有关的一个例子是线性规划问题. 令 A 为 $m \times n$ 的整数矩阵，d 和 c 分别为长度 m 和 n 的整数向量，k 为整常数. 问是否存在长度 n 的有理向量 x，同时满足：

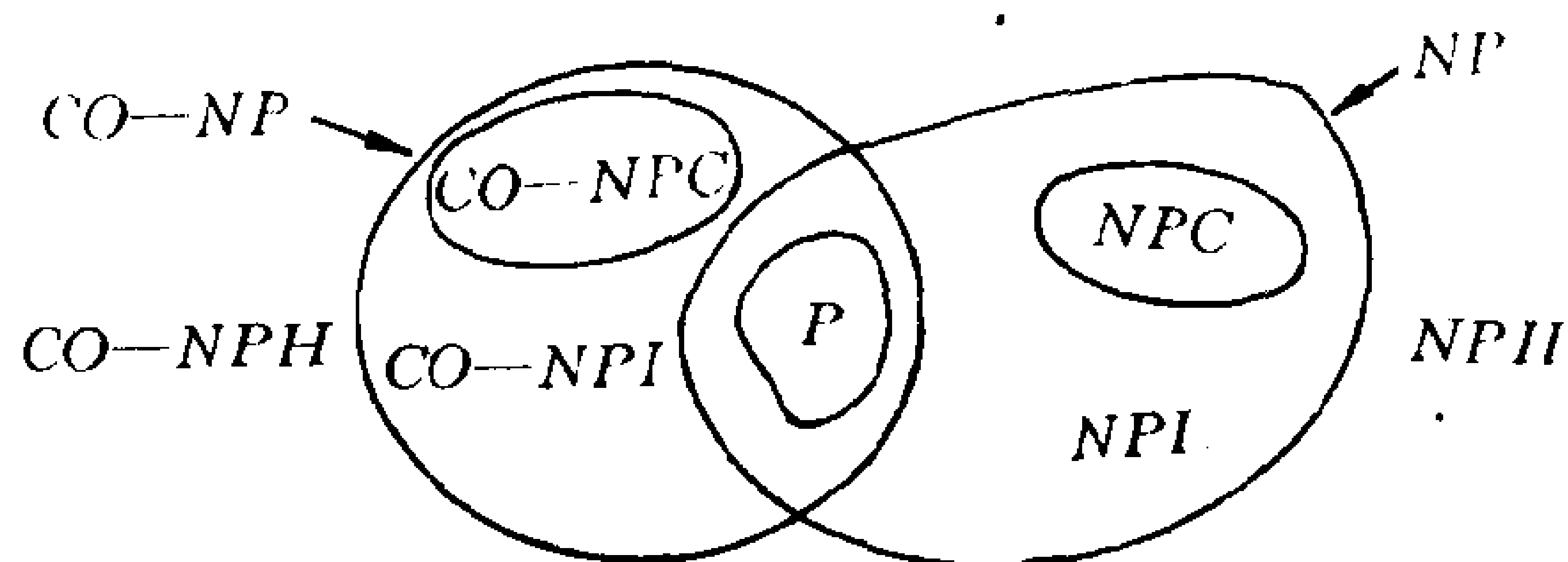
$$Ax' \leqslant d' \quad \text{和} \quad cx' \geqslant k. \tag{11.4}$$

这个问题原来不知道它是否为指数型. 1979 年苏联数学家哈吉扬证明了它属于 P 类，引起国际数学界的重视. 实际上人们早已知道它和它的补问题都属于 NP 类. 根据上面的推论，它不属于 NPC 类是意料中的事，不值得奇怪.

那么，NPC 类的问题是否就是最难最难的问题了呢？不一定. 如果 NPC 类的问题可以归约到某个问题 R（此处归约的概念有所推广，不详述），而 R 未能归约到 NPC 类中的任何问题，则 R 称为是具有 NP 难度的. 我们称这类问题为 NPH 类. 其中一个问题如下：

设命题公式 E 由命题变量集 V、常量 T 和 F、以及操作符 ∧、∨、～、→等组成. 任给非负整数 $K>0$，问是否存在长度不超过 K 的另一命题公式 E'，使 $E' \equiv E$？

我们当然希望将来能证明 NPH 是空集，不过相反的可能性也存在. 如果用一个图表示，那么以上提到的各问题类的相互关系大致如下：



比起数学中许多古老的难题来,P=NP？问题要年轻得多.然而看来它的难度并不亚于某些古老的难题.有人甚至说,一旦P=NP？问题解决了,全世界的数学家可以放三天假以示庆祝.

第十二章　最小不动点理论

——解开递归迷雾的钥匙

我们在第三章里已经讨论过递归函数的理论.在这一章里,我们要具体研究一下解递归函数的方法.函数的递归定义可以看成是函数方程,复杂的函数方程的解不是一眼就能看出来的,各种意想不到的情况都可能出现,让我们从最简单的情况开始,举几个例子.其中大写字母 F 表示函数变项,即待定的函数.

$$F(x)\equiv F(x). \tag{12.1}$$

形式上,这也是一个函数方程,但它没有对候选函数施加任何限制,因之也就没有提供任何信息.所有函数都是该方程的解.

$$F(x)\equiv F(x)+1, \tag{12.2}$$

这是一个矛盾的函数方程,任何有意义的函数都不可能是这个方程的解.

$$F(x)\equiv \text{if} x=0 \text{ then } 1 \text{ else } F(x-1) \text{ fi}. \tag{12.3}$$

在上面的方程中,显然 $F(x)$ 对所有的非负整数有唯一定义,即

$$F(n)=1, \qquad \text{对所有 } n\geqslant 0. \tag{12.4}$$

但是，当 x 为负数时，$F(x)$的值就不确定了. 下列任一函数 $f_i(n)$都是函数方程(12.3)的解：

$$f_i(n)\begin{cases}=1, & \text{对所有 } n\geqslant 0\\ =i, & \text{对所有 } n<0\end{cases} \tag{12.5}$$

其中 i 可为一任意整数.

再看下面的方程：

$$F(x)\equiv \text{if} x > 100 \text{ then } x-10 \quad \text{else } F(F(x+11)) \text{ fi} \tag{12.6}$$

这个方程的解就更难一眼看出来了，因为定义部分有一个 F 嵌套 $F(F(x+11))$. 实际上，该方程有一个很简单的解：

$$f(x)\equiv \text{if } x>100 \text{ then } x-10 \text{ else } 91 \quad \text{fi} \tag{12.7}$$

读者不妨自己验证或推导一下这个解. 它有个名字，叫麦卡锡 91 函数，因为当 $x\leqslant 100$ 时，函数的值都是 91.

再看一个更复杂的例子：

$$F(x,y)\equiv \text{if } x$$
$$=y \text{ then } y+1 \text{ else } F(x,F(x-1,y+1)) \text{ fi} \tag{12.8}$$

这里有两个变元，函数还带嵌套. 该函数方程至少有下列三个解：

$$\begin{aligned} &f_1(x,y)\equiv \text{if } x=y \text{ then } y+1 \text{ else } x+1,\\ &f_2(x,y)\equiv \text{if } x\geqslant y \text{ then } x+1 \text{ else } y-1,\\ &f_3(x,y)\equiv \text{if } (x\geqslant y)\wedge(x-y \text{ 为偶数})\\ &\qquad\qquad \text{then } x+1 \text{ else undefined fi.}\end{aligned} \tag{12.9}$$

这里 undefined 表示无定义.

看到这里，读者心中可能会升起一种感觉：解递归定义的函数方程是很复杂的事，我们必须有一套严密的理论和方法来做这件事. 本章的内容就是论述这个问题的.

我们从式(12.3)开始来阐述解递归方程的基本思想. 前面

已经见到，对任一 i，式(12.5)中的 $f_i(n)$ 必是式(12.3)中函数方程的解，于是我们可以提这样的问题：取哪一个 $f_i(n)$ 作为函数方程的解才最合理呢？回答是：它们之中哪一个也不是(12.3)的合理解，因为函数方程只对 $F(n)$ 在 $n \geqslant 0$ 处的值作了规定，而对所有的 $n<0$ 处的 $F(n)$ 的值却没有提供信息. (12.5)规定在 $n<0$ 处 $f_i(n)=i$，实际上是人为地加进了原来的函数方程所没有的信息. 这种信息完全是随意的，因而是无意义的，方程(12.3)的合理解应该是：

$$\begin{aligned} &f(n)=1, && n \geqslant 0. \\ &f(n)\text{无定义}, && n<0. \end{aligned} \tag{12.10}$$

用"无定义"来表示 $f(n)$ 在 $n<0$ 处的值还可以从另一个角度来解释：如果把函数方程(12.3)看成一个程序，则此程序对 x 的任何非负值都能在作有限步计算后停止并给出 $f(x)$ 的值. 其方法是沿整数轴一步步地后退，直至退到 $x_0=0$，就可利用 $f(x_0)=1$ 的事实来给出 $f(x)$ 的值. 但是，对小于 0 的 x 值，$f(x)$ 的计算不可能终止. 它首先试图利用 $f(x-1)$ 的值，接着又试图利用 $f(x-2)$ 的值，……，这个过程永远不会结束. 因此我们称 $f(x)$ 在 $x<0$ 处的行为是发散的. 这种发散的函数值，当然是无定义的.

通过刚才的讨论，我们得到了两条重要的原则：

1. 引进无定义作为一种可能的函数值.

2. 如果函数方程有两个可能的解函数在某处的值不一样，则我们只接受在该处取无定义值的解函数.

利用上面的第一条原则，我们立即求出了式(12.2)中矛盾方程的解，这是一个处处取无定义值的函数. 利用上面的第二条原则，我们又可立即推得式(12.1)中函数方程的解也是处处无定义函数，这与直观的印象(任何函数都是该方程的解)很不一

样，可能会出乎一些读者的意料之外. 至于式(12.8)中的函数方程，则显然只有 $f_3(x,y)$ 才是它的解，因为该函数方程对两种情况未提供任何信息. 第一，$x<y$，第二，$x>y$ 且 $x-y$ 为奇数，对这两种情况，递归求解会无休无止地继续下去，实际上是函数取无定义值的部份.

从现在开始，我们用符号 $\perp$ 表示无定义值. 由于函数可以嵌套，如 $g(f(x))$，当对某个 x_0，$f(x_0)=\perp$ 时，函数 g 不可避免地要面临变元为 $\perp$ 的情况. 因此，我们干脆把 $\perp$ 加进函数的定义域中去，如果原来的域是 D，则以 $D^{\perp}=D\cup\{\perp\}$ 表示经过扩充无定义元素以后的域. 通常约定，对任何函数 g 和任何 x_i，$g(x_1,\cdots,\perp,\cdots,x_n)=\perp$. 这称为 g 的自然扩充.

由于我们在下面要用逼近(或称叠代)的方法来解函数方程，因此需要某种方向或序的概念. 这里要的不是全序，而是偏序.

设 D 是由一些元素构成的集合，D 上的偏序 $\sqsubseteq$ 是 D 的元素间的一种二元关系，它满足如下三个规定：

1. 自反律：$a\sqsubseteq a$.

2. 传递律：由 $a\sqsubseteq b$ 及 $b\sqsubseteq c$ 可得 $a\sqsubseteq c$.

3. 恒等律：由 $a\sqsubseteq b$ 及 $b\sqsubseteq a$ 可得 $a=b$

由于需要不同，不同的文献对偏序的定义可能在第 3 点上有出入. 例如有的作者排除了 $a\sqsubseteq b$ 和 $b\sqsubseteq a$ 同时成立的情况.

偏序的例子很多. 如果我们用 $A\sqsubseteq B$ 表示集合 B 包含集合 A，则一个集合中各子集的相互包含关系定义了该集合上的一个偏序. 又如，一树形结构中上级节点和下级节点之间的关系也定义了该树形结构上的一个偏序. 此外，全序也是偏序的一个特例，一个偏序称为全序，当且仅当对任意两个元素 a 和 b，$a\sqsubseteq b$ 和 $b\sqsubseteq a$ 这两个关系至少有一个成立.

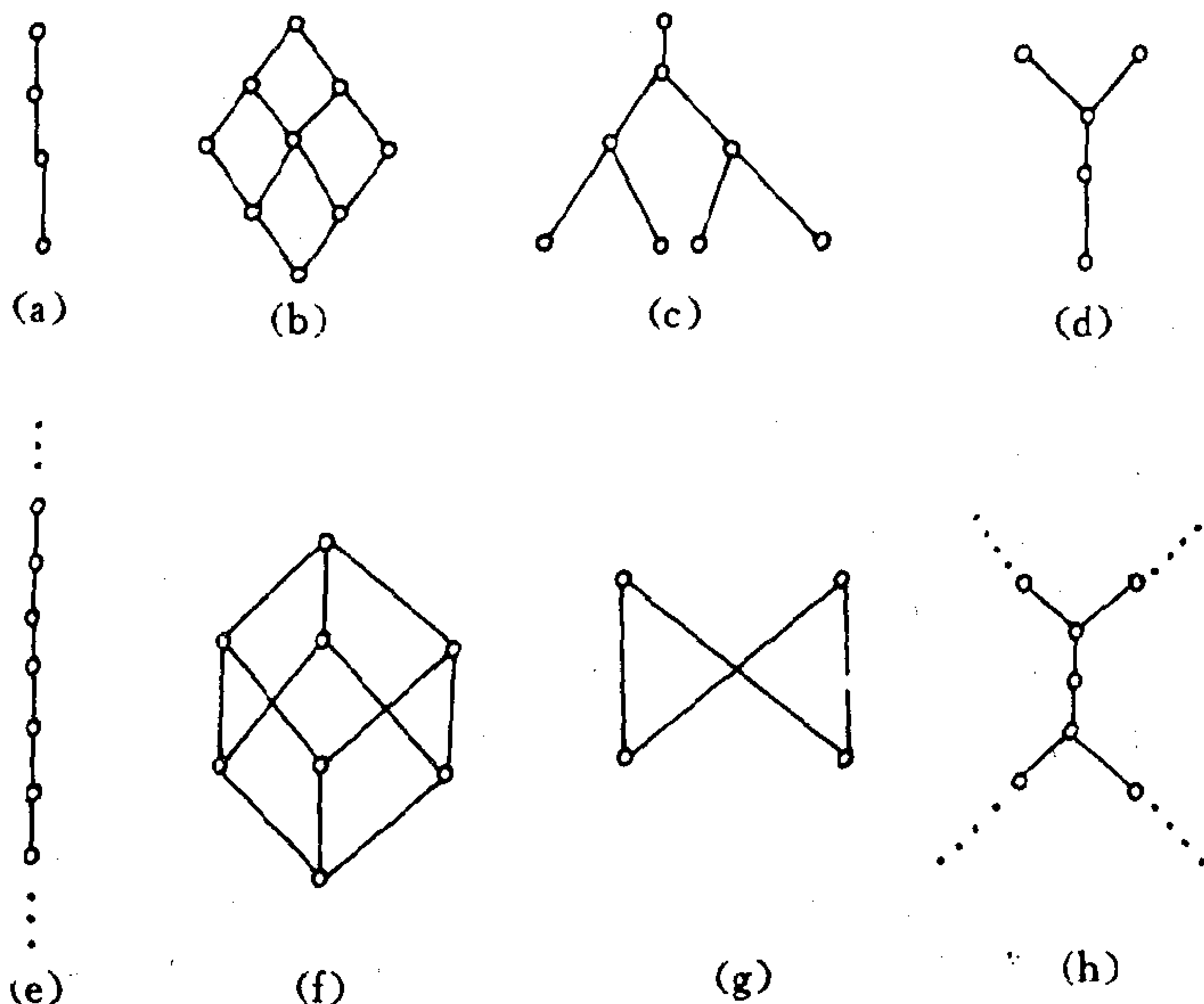
我们用一个形象的说法：a 小于等于 b，来表示 $a \sqsubseteq b$，并且用 $a \sqsubset b$ 表示 $a \sqsubseteq b$ 且 $a \neq b$. 如果偏序集 P 的元素 a 小于等于 P 中的所有元素，则称 a 为 P 的最小元素. 反之，如果 P 的所有元素都小于等于 a，则称 a 为 P 的最大元素. 很容易证明，如果偏序集的最小元素和最大元素存在（它们不一定存在！），则一定是唯一的.

如果对偏序集 P 的元素 a，不存在另一个元素 x，使 $x \sqsubseteq a$ 且 $x \neq a$，则 a 称为是 P 的一个极小元素. 类似地可以定义 P 的极大元素（应该怎么定义？请考虑）. 不难证明，对任何偏序集，极小元素和极大元素一定存在，但不一定唯一.

设 M 是偏序集 P 的一个子集，如果 P 的元素 a 小于等于 M 中的所有元素，则称 a 是 M 的一个下界. 如果 M 的下界（它不一定唯一）集合有一最大元素 b，则 b 称为 M 的下确界. 类似地可以定义 M 的上界和上确界. 注意 M 的下确界和上确界不一定存在. 如果它们存在，则分别以 $\sqcup M$ 和 $\sqcap M$ 表之.

122 面的图给出了一些偏序集合的例子，图中以小圆圈表示偏序集中的元素，以元素位置高低和联线表示偏序. 若 b 高于 a 且 b 有连线到 a，则表示 $a \sqsubseteq b$. 其中 (a) 和 (e) 是全序，(e) 和 (h) 是无限偏序集. (a)、(b) 和 (f) 有最大和最小元素，(c) 只有最大元素，没有最小元素，(d) 只有最小元素，没有最大元素. (e)、(g) 和 (h) 既没有最小元素，又没有最大元素. (c) 有四个极小元素，(g) 有两个极小元素和极大元素. (c) 的极小元素集有上确界，(g) 的极小元素集没有上确界，它的极大元素集也没有下确界. (a)、(c)、(d) 可以看作（有限的）树形结构，(e) 和 (h) 可以看作（无限的）树形结构. (b)、(f) 和 (g) 不是树形结构.

在偏序家族中有一个特殊的成员，叫做格，它是一个重要的数学概念，有着广泛的应用. 我们用前面引进的两个特殊运算 $\sqcup$



和⊓来定义格，它们的含义仍然是上确界和下确界，但在格中它们首先是二元运算. 规定：

1. 若 a,b 属于格 L，则 $a\sqcup b$ 和 $a\sqcap b$ 也属于格 L.

2. 交换律：$a\sqcap b=b\sqcap a, a\sqcup b=b\sqcup a$. (12.11)

3. 结合律：

$$(a\sqcap b)\sqcap c=a\sqcap(b\sqcap c),$$
$$(a\sqcup b)\sqcup c=a\sqcup(b\sqcup c). \quad (12.12)$$

4. 恢复律：

$$a\sqcap(a\sqcup b)=a,\quad a\sqcup(a\sqcap b)=a. \quad (12.13)$$

根据交换律和结合律，可以知道像 $\sqcup M$ 和 $\sqcap M$ 这样的写法是有意义的，其中 M 是格 L 中的一个任意有限集合，具体来说，当 $M=\{a_1,a_2,\cdots,a_n\}$ 时，

$$\bigsqcup M = a_1 \sqcup a_2 \sqcup \cdots\cdots \sqcup a_n,$$
$$\bigsqcap M = a_1 \sqcap a_2 \sqcap \cdots\cdots \sqcap a_n. \quad (12.14)$$

如果当 M 是无限集时，上面的这些性质还成立，则称格 L 是一个完全格.

实际上，格上的关系 $\sqsubseteq$ 和两个运算 $\sqcup$ 及 $\sqcap$ 是抽象的关系和抽象的运算. 所谓"小于等于"，"上确界"，"下确界"之类的提法不过是人为地赋予它们一个直观的解释，如果给别的解释同样也是可以的，运算 $\sqcup$ 和 $\sqcap$ 是完全对称的，如果 A 是有关格的一个定理，则把 A 中的 $\sqcup$ 换成 $\sqcap$，$\sqcap$ 换成 $\sqcup$，得到的仍是一个定理，这叫做对偶原理.

格有许多有趣的性质，例如，利用运算 $\sqcup$ 和 $\sqcap$ 的性质，可以证明 $a \sqcap a = a, a \sqcup a = a$（这叫零幂律）. 证明过程如下：

令 $b = a \sqcap a$，则

$$a = a \sqcap (a \sqcup b) = a \sqcap (a \sqcup (a \sqcap a))$$
$$= a \sqcap (a) = a \sqcap a,$$

利用对偶原理立得 $a = a \sqcup a$.

作为练习，建议读者自己证明下面的两个定理：

定理 12.1 设 a, b 是格中的元素，则有：

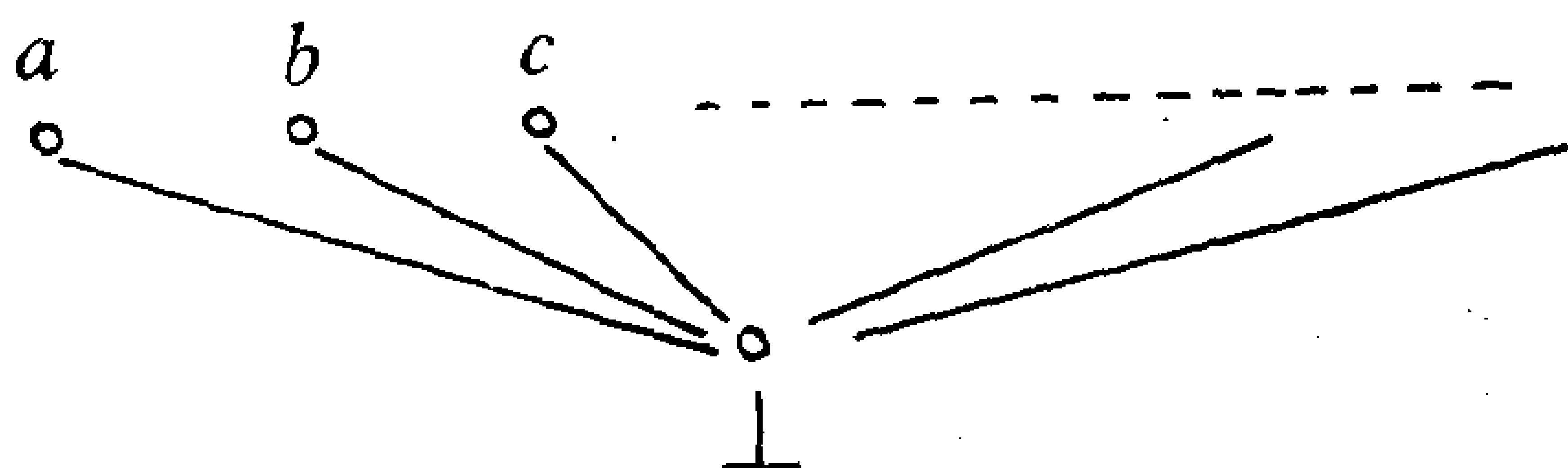
$a \sqcap b = a$ 当且仅当 $a \sqcup b = b$.

定理 12.2 设 a、b 是格中的元素，则由 $a \sqcap b = a \sqcup b$ 可推出 $a = b$.

为了研究递归函数的解，我们在上面用了比较多的篇幅介绍偏序和格这两个概念. 现在言归正传，要利用这两个概念展开一套解递归函数的理论. 首先，我们在前面已定义过的扩域 $D^{\perp}$ 上引进偏序如下：若 $a, b \in D^{\perp}$，则 $a \sqsubseteq b$，当且仅当 $a = b$ 或 $a = \perp$.

由此可见，$D^{\perp}$ 上的偏序只有两层. 上面是并列的诸普通元

素，下面是一个无定义元素⊥，如下图所示. 这种偏序常称为平坦偏序.



$D^{\perp}$只能表示一个变元的定义域，如果函数有多个变元怎么办？可以引进 n 个定义域的拓扑积，令

$$D^{n\perp}=D_1^{\perp}\times D_2^{\perp}\times\cdots\cdots\times D_n^{\perp}, \tag{12.15}$$

其中每个 $D_i^{\perp}$ 表示函数的第 i 个变元的定义域经扩充无定义元素⊥后所得的域. $D^{n\perp}$上的偏序可以定义为：

$$\langle a_1,\cdots,a_n\rangle\sqsubseteq\langle b_1,\cdots,b_n\rangle, \tag{12.16}$$

当且仅当对所有的 i，有 $a_i\sqsubseteq b_i$.

我们给了符号⊥以双重意义，它既是无定义元素，又是偏序中的最小元素(如果存在)，这样的双重意义不会带来矛盾. 事实上，这已是通用的符号. 偏序集的最大元素(如果存在)，则常以符号⊤表示，注意不要把它和英文字母 T 混淆了.

函数本身也可以构成偏序. 设 L_1 和 L_2 为格，f 和 g 都是在 L_1 上定义，把 L_1 映入 L_2 的函数，则当且仅当对所有 x，$f(x)\sqsubseteq g(x)$时，认为 f"小于等于"g，用 $f\sqsubseteq g$ 表示. 由函数构成的偏序集都有一个最小元素，即处处无定义函数 Ω，对任何 x 有，$\Omega(x)=\perp$. 当然，这要求格 L_2 必须有最小元素⊥.

扩域 $D^{\perp}$并不是格，因为对任意 a、b，它们的上确界 $a\sqcup b$ 不一定存在(下确界恒有). 但我们总可以设想 $D^{\perp}$也有最大元素⊤，并且永远有 $f(\top)=\top$，且对任何 $a\neq\top$，$f(a)\neq\top$. 所以把 $D^{\perp}$当作格来讨论并无问题.

一个简单的推论是：若 f、g 都是把 $D_1^\perp$ 映入 $D_2^\perp$ 的函数. 凡 f 和 g 均有定义之处(即它们的值不为 $\perp$ 之处)，f 和 g 的值相等. g 没有定义之处，f 也没有定义，则 $f \sqsubseteq g$. 如果回忆一下我们在本节开头时给出的解递归函数的第二条原则，则可以粗略地说，我们要找的是所有可能的函数解中"最小的"那个解. 我们在下面会在更严格的意义上理解这句话的含义.

设 f 是把格 L_1 映入格 L_2 的函数，则当且仅当对所有的 $x \sqsubseteq y$ 均有 $f(x) \sqsubseteq f(y)$ 时，称函数 f 为单调的. 打个比方来说，每年的电视剧评选要选出最佳电视剧和最差电视剧，另外还要评一些单项奖，则评比结果对参赛电视剧构成一个偏序，甚至是一个格，如果发的奖品也构成格的话(未评上的电视剧发纪念奖，最差电视剧不给奖)，则发奖办法就是格上的函数，一个公平的发奖办法就是格上的单调函数.

一般地说，我们只对单调的函数感兴趣. 从直观上说，$x \sqsubseteq y$ 表示 y 包含的信息不少于 x 包含的信息(例如，若 $f(x)=\perp$ 而 $g(x)\neq\perp$，则 $g(x)$ 包含比 $f(x)$ 更多的信息). 在单调函数的情况下，如果变元包含的信息多，则函数值包含的信息也多，这是合乎逻辑的.

举几个简单的例子. 恒等函数 $f(x)=x$，常数函数 $f(x)=a$ 和处处无定义函数 $\Omega(x)\equiv\perp$ 都是单调函数. 实际上，单调函数的类是很大的，这由下面的定理可以看出：

定理 12.3 设 f 是 $D^{n\perp}$ 上经过自然扩充的函数，其中 $D^{n\perp}=D_1^\perp\times D_2^\perp\times\cdots\times D_n^\perp$，则 f 是单调的.

证：如果 f 不单调，则必有变元组 $\langle a_1,\cdots,a_n\rangle$ 及 $\langle b_1,\cdots,b_n\rangle$，使 $\langle a_1,\cdots,a_n\rangle \sqsubseteq \langle b_1,\cdots,b_n\rangle$，但 $f(a_1,\cdots,a_n) \sqsubseteq f(b_1,\cdots,b_n)$ 不成立. 根据定义知道对每个 i 有 $a_i \sqsubseteq b_i$. 如果 a_i 都等于 b_i，则 $f(a_1,\cdots,a_n)=f(b_1,\cdots,b_n)$，与假设矛盾. 因此至少有一个 $a_i\neq b_i$，此 a_i

只能是$\perp$.根据自然扩充的定义知$f(a_1,\cdots,a_n)=\perp$,因此必有$f(a_1,\cdots,a_n)\sqsubseteq f(b_1,\cdots,b_n)$,矛盾.

定理 12.4 单调函数的组合仍是单调函数.即,若$f_1,f_2,\cdots,f_n$及n秩函数g都是单调的,则$g(f_1(\cdots),\cdots,f_n(\cdots))$也是单调的.

这个定理的证明很简单,我们留给读者.

一个偏序集(或格)L的全序子集M称为L中的一个链.链$\{a_j\}$中满足$a_i\sqsubset a_{i+1}$的元素a_i的个数称为链$\{a_j\}$的长度.函数也可以构成链.设$f_i,i=0,1,2,\cdots$,都是D_1到D_2的单调函数,满足$f_0\sqsubseteq f_1\sqsubseteq f_2\sqsubseteq\cdots$,则序列$\{f_i\}$称为$D_1$到$D_2$的一个函数(映射)链.如果$f$也是$D_1$到$D_2$的连续函数,并且对任意$i$有$f_i\sqsubseteq f$,则$f$称为链$\{f_i\}$的上界.若对$f_i$的任意上界$g$,均有$f\sqsubseteq g$,则称$f$为$\{f_i\}$的上确界.

什么时候$\{f_i\}$的上确界存在呢?为了说明这一点,需要引进链完备的新概念.如果偏序集D中的任意一个链$\{a_i\}$都有一个上确界,则称D是链完备的.于是我们有:

定理 12.5 设函数序列$\{f_i\}$中的每个f_i都是把D_1映入D_2的单调函数.并且D_1和D_2都是链完备的,则如果$\{f_i\}$是一个链,一定有上确界f.

本定理的证明是不难的,它的大意如下:利用链完备的性质,证明对每个a,链$\{f_i(a)\}$的上确界a'存在.然后利用$a\rightarrow a'$的对应关系定义新函数f.利用诸f_i的单调性,证明f也是单调的,最后从f的构造过程可以看出f确是$\{f_i\}$的上确界.

任何扩域D^+都是链完备的.任何有限格(只有有限多元素的格)和有限层次格(存在只与格有关的常数N,使任意链之长不超过N)以及完全格都是链完备的.

现在要讲到一个最最关键的概念,就是不动点.设函数f

是把偏序集 D 映入 D 的函数. 如果存在 D 中元素 x,使 $f(x)=x$,则 x 称为 f 的不动点. 不动点概念的重要性在于:求得了函数 f 的不动点也就是求得了函数方程 $f(x)=x$ 的解.

关于不动点,塔尔斯基有一个很漂亮的定理:若 L 是完全格,则每个在 L 上定义并在 L 上取值的单调函数都有一个非空的不动点集,该集也构成一个完全格.

不过我们真正关心的,还不是一个函数的不动点,而是由函数方程定义的泛函的不动点. 回忆一下本章开头讲到的那些函数方程,它们实际上是定义了一些如下形式的泛函:

$$F=\tau[F]. \tag{12.17}$$

粗略地说,泛函就是函数的函数. 说得严格一点:设 F 是由全体把格 L_1 映入格 L_2 的单调函数构成的集合. 如果 τ 是由 F 到 F 的映射,则称 τ 是一个泛函. 为了解函数方程,我们真正需要的,是求泛函的不动点.

像对函数一样,对泛函也可以定义一些良好的性质. 我们规定:如果由 $f\sqsubseteq g$ 总可以推出 $\tau[f]\sqsubseteq\tau[g]$,则称 τ 为单调泛函,这里 f 和 g 都是 F 中的函数. 此外,如果 τ 是单调泛函,并且对 F 中的任何链 $\{f_i\}$ 有

$$\tau[\mathrm{lub}\{f_i\}]=\mathrm{lub}\{\tau[f_i]\}, \tag{12.18}$$

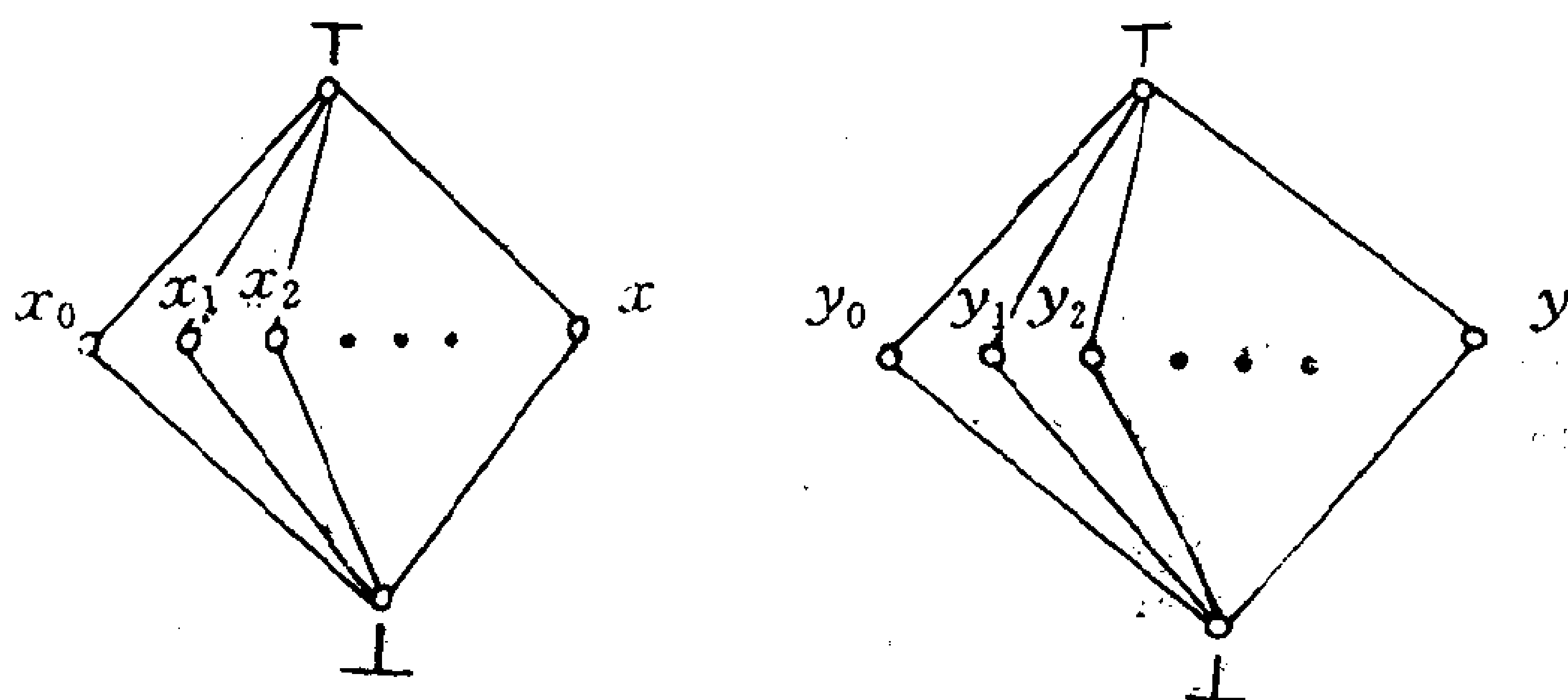
则称 τ 是连续泛函,其中 lub 表示上确界. 这里连续的直观含义是:泛函的计算和求上确界的计算(这是一个极限过程)可以交换次序.

可以看出,连续泛函要求函数的值域 D_2 是链完备的,否则上确界 $\mathrm{lub}\{f_i\}$ 不一定存在. 另一方面,只要 D_2 是链完备的,$\mathrm{lub}\{f_i\}$ 和 $\mathrm{lub}\{\tau[f_i]\}$ 一定存在,因为由 $\{f_i\}$ 是一个链可以推出 $\{\tau[f_i]\}$ 也是一个链.

举几个简单的连续泛函的例子. 恒等泛函 $\tau[f]=f$ 是连续

的. 常数泛函 $\tau[f]=h$ 也是连续的.

再举一个不是连续泛函的例子. 在下面的图中, $\{x_i\}$ 和 $\{y_i\}$ 都是无穷序列, $i=0,1,2,\cdots$. 函数 f_i 把 $x_0,x_1,\cdots,x_i$ 映为 $y_0,y_1,\cdots,y_i$, 其它的 $x_j, j>i$, 皆映为 $\perp$. 易见 $\text{lub} f_i=f$, f 把所有 x_k 映为 $y_k, k=0,1,2,\cdots$. 现在令泛函 τ 把所有 f_i 映为处处无定义函数 Ω, 而把 f 映为 f, 则 τ 是单调的, 但不连续, 因为 $\text{lub}\{\tau[f_i]\}=\Omega$, 而 $\tau[\text{lub}\{f_i\}]=f$.



那么, 一个泛函满足什么条件就可以成为连续泛函呢? 对此有如下的定理:

定理 12.6 对有限层次格, 由单调函数和函数变量 F 组合而成的泛函是连续的.

证: 用归纳法. 设此泛函为 τ. 当 τ 仅由一个函数变量 F 组成, 或 τ 是一个固定的单调函数时, 容易看出 τ 肯定是连续的.

以 τ 中所含函数变量 F 和单调函数调用的数量作为 τ 的结构复杂度. 我们设法对任意的 τ 降低这一复杂度.

情况一: τ 的最外层是一单调函数, 即

$$\tau[F]\equiv f(\tau_1[F],\cdots,\tau_n[F]), \tag{12.19}$$

其中 f 是单调函数. 根据归纳假设, 诸 τ_i 都是连续的. 由 τ_i 的单

调性知对任意的 $g \sqsubseteq h$ 可以推出 $\tau_i[g] \sqsubseteq \tau_i[h]$. 由 f 的单调性知 $f(\tau_1[g],\cdots,\tau_n[g]) \sqsubseteq f(\tau_1[h]),\cdots,\tau_n[h])$，即 $\tau[g] \sqsubseteq \tau[h])$，说明 τ 是单调泛函.

现设 $\{f_i\}$ 是一条链. 根据链完备的假设，其上确界 $\mathrm{lub}\{f_i\}$ 存在，且对每个 i 有 $f_i \sqsubseteq \mathrm{lub}\{f_i\}$. 由诸 τ_i 及 f 的单调性知 $\tau[f_i] \sqsubseteq \tau[\mathrm{lub}\{f_i\}]$. 另一方面，$\{\tau[f_i]\}$ 也是一个链，其上确界 $\mathrm{lub}\{\tau[f_i]\}$ 也存在. 由上确界的定义知：

$$\mathrm{lub}\{\tau[f_i]\} \sqsubseteq \tau[\mathrm{lub}\{f_i\}]. \tag{12.20}$$

另一方面，令 t 为格中任一元素，我们有

$$\begin{aligned}&\tau[\mathrm{lub}\{f_i\}](t)\\ &\quad = f(\tau_1[\mathrm{lub}\{f_i\}](t),\cdots,\tau_n[\mathrm{lub}\{f_i\}](t))\\ &\quad = f(\mathrm{lub}\{\tau_1[f_i]\}(t),\cdots,\mathrm{lub}\{\tau_n[f_i]\}(t)),\end{aligned}$$

其中最后一个“＝”符号基于归纳假设.

由于格是有限层次的，因此必存在正整数 I，使 $i>I$ 时，所有 $\tau_j[f_i](t)=\tau_j[f_I](t)$，于是我们有

$$\begin{aligned}&\tau[\mathrm{lub}\{f_i\}](t)\\ &= f(\tau_1[f_I](t),\tau_2[f_I](t),\cdots,\tau_n[f_I](t))\\ &= \tau[f_I](t) \sqsubseteq \mathrm{lub}\{\tau[f_i]\}(t).\end{aligned} \tag{12.21}$$

比较(12.20)和(12.21)，可知 τ 是连续泛函.

情况二：τ 的最外层是函数变量名 F，即

$$\tau[F] \equiv F(\tau_1[F],\tau_2[F],\cdots,\tau_n[F]). \tag{12.22}$$

证法类似于情况一，只需以 F 取代 f 即可.

本定理原则上也适用于扩域 D^+，因为我们在前面已经说过，无妨假设扩域有一个最大元素 $\top$，从而构成有限层次格.

下面引进泛函不动点的概念. 以 F 表示全体把格 L_1 映入格 L_2 的单调函数的集合. τ 为泛函 $F \to F$. 如有 F 中函数 f，使 $\tau[f]=f$，则称 f 为 τ 的不动点. 如果 f 是 τ 的不动点，并且对 τ

的其它不动点 g 均有 $f \sqsubseteq g$，则 f 称为 τ 的一个最小不动点. 最小不动点是递归函数求解的核心概念. 只有当某个函数 f 是函数方程(泛函)的最小不动点时，才承认 f 是函数方程的解.

由最小不动点的定义立即看出：任何泛函 τ 至多有一个最小不动点. 问题是在什么条件下最小不动点存在呢？

定理 12.7 任何连续泛函 τ 均有一个最小不动点.

证：由于 τ 是连续的，它必须是单调的. 我们用逐步逼近的方法，并用处处无定义函数 Ω 作为逼近的起点. 函数序列

$$\Omega, \tau[\Omega], \tau^2[\Omega], \cdots, \qquad (12.23)$$

构成一个链，它的上确界 tau 必须存在(参看前面引进连续泛函概念时的说明).

tau 是一个不动点，这是因为：

$$\tau[\mathrm{tau}] \equiv \tau[\mathrm{lub}\{\tau^i[\Omega]\}]$$

$$\equiv \mathrm{lub}\{\tau^{i+1}[\Omega]\} \quad \text{(根据 } \tau \text{ 的连续性)}$$

$$\equiv \mathrm{tau}.$$

其次，tau 是一个最小不动点. 因为假设 τ 有另一个不动点 g，则由于

$$\Omega \sqsubseteq g, \qquad (12.24)$$

$\forall\ i, \tau^i[\Omega] \sqsubseteq \tau^i[g] = g.$

说明 g 是诸 $\{\tau^i[\Omega]\}$ 的上界，但因 tau 是 $\{\tau^i[\Omega]\}$ 的上确界，所以必有 tau $\sqsubseteq$ g.

请注意，这个定理是本章中最重要的定理，它不但从理论上论证了最小不动点存在的条件，而且给出了构造最小不动点的一般性方法. 现在我们要利用这个定理求本章开始时列出的那些递归函数的解. 注意到定理 12.6，我们只需证明由它们定义的泛函都是单调泛函就可以了. 又注意到第 129 页上提到的一个简单的推论，这个证明过程可以通过如下两条简单的证明链

来完成. 设 f 和 g 都是 $D_1^+ \to D_2^+$ 的单调函数，且 $f \sqsubseteq g$，则如下推导成立：

对任意 $x \in D_1^+$：

1. $\tau[f](x) \neq \bot$ 且

$\tau[g](x) \neq \bot \Longrightarrow f(x) \neq \bot$ 且 $g(x) \neq \bot$

$\Longrightarrow f(x) = g(x) \Longrightarrow \tau[f](x) = \tau[g](x)$.

2. $\tau[g](x) = \bot \Longrightarrow g(x) = \bot \Longrightarrow f(x) = \bot$

$\Longrightarrow \tau[f](x) = \bot$.

本章开头的几个泛函全部符合上述两条检验，读者不妨自己试证一下. 因此，我们可以放心大胆地使用定理 12.7 的构造方法.

1. 对式(12.1)，注意对所有 i，$\tau^i[\Omega] = \Omega$，这证实了我们在第 119 页上的断言：该方程的（最小不动点）解是处处无定义函数.

2. 对式(12.2)有同样结果. 因为我们早已规定，对任何函数 f，$f(\cdots, \bot, \cdots) = \bot$.

3. 对式(12.3)，我们有

$\tau^i[\Omega] \equiv f(x) = 1$，　　当 $x = 0, 1, \cdots, i-1$，

$f(x) = \bot$，　　其它.

因此最小不动点解是：

$\mathrm{lub}\{\tau^i[\Omega]\} \equiv f(x) = 1$，　x 为非负整数，

$f(x) = \bot$，　其它.

4. 对式(12.6)，我们有：

当 $1 \leqslant i \leqslant 11$ 时，

$\tau^i[\Omega](x) \equiv$ if $x > 100$ then $x - 10$

else if $x > 101 - i$ then 91 else $\bot$ fi fi

当 $i \geqslant 11$ 时，

$\tau^i[\Omega](x) \equiv$ if $x>100$ then $x-10$ else

if $x>90-11(i-11)$ then 91 else $\perp$ fi fi

不难看出它的最小不动点解是

$\text{lub}\{\tau^i[\Omega]\}(x) \equiv$ if $x>100$ then $x-10$ else 91 fi

与前面讲的一致.

5. 对式(12.8),我们在式(12.9)中给出的三个函数都是它的不动点,但其中只有 f_3 是最小不动点,读者可以自己验证一下.